新世纪高职高专教改项目成果教材

软 件 测 试

赵瑞莲　编

高等教育出版社

内容提要

本书是教育部新世纪高职高专教育人才培养模式和教学内容体系改革与建设项目成果，是组织有关教育部高职高专教育专业教学改革试点院校编写的。

主要内容包括 绪论、软件测试实质、软件测试策略、黑盒测试、白盒测试、集成测试与系统测试，验证测试和确认测试，测试计划与测试文档、面向对象的软件测试。

本书适合于高等职业学校、高等专科学校、成人高校、示范性软件职业技术学院、本科院校及其举办的二级职业技术学院、继续教育学院以及民办高校使用，也可供计算机专业人员和爱好者参考使用。

图书在版编目(CIP)数据

软件测试/赵瑞莲编. —北京：高等教育出版社，
2004.1
ISBN 7-04-013698-8

Ⅰ.软… Ⅱ.赵… Ⅲ.软件-测试-高等教育-教材 Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字(2003)第 116467 号

出版发行 高等教育出版社
社　　址 北京市西城区德外大街 4 号
邮政编码 100011
总　　机 010-82028899

购书热线 010-64054588
免费咨询 800-810-0598
网　　址 http://www.hep.edu.cn
http://www.hep.com.cn

经　　销 新华书店北京发行所
排　　版 高等教育出版社照排中心
印　　刷

开　　本 787×1092 1/16
印　　张 14.25
字　　数 340 000

版　　次 年 月第 1 版
印　　次 年 月第 次印刷
定　　价 18.20 元

出 版 说 明

为认真贯彻《中共中央国务院关于深化教育改革全面推进素质教育的决定》和《面向21世纪教育振兴行动计划》,研究高职高专教育跨世纪发展战略和改革措施,整体推进高职高专教学改革,教育部决定组织实施《新世纪高职高专教育人才培养模式和教学内容体系改革与建设项目计划》(教高[2000]3号,以下简称《计划》)。《计划》的目标是:"经过五年的努力,初步形成适应社会主义现代化建设需要的具有中国特色的高职高专教育人才培养模式和教学内容体系。"《计划》的研究项目涉及高职高专教育的地位、作用、性质、培养目标、培养模式、教学内容与课程体系、教学方法与手段、教学管理等诸多方面,重点是人才培养模式的改革和教学内容体系的改革,先导是教育思想的改革和教育观念的转变。与此同时,为了贯彻落实《教育部关于加强高职高专教育人才培养工作的意见》(教高[2002]2号)的精神,教育部高等教育司决定从2000年起,在全国各省市的高等职业学校、高等专科学校、成人高等学校以及本科院校的职业技术学院(以下简称高职高专院校)中广泛开展专业教学改革试点工作,目标是:在全国高职高专院校中,遴选若干专业点,进行以提高人才培养质量为目的、人才培养模式改革与创新为主题的专业教学改革试点,经过几年的努力,力争在全国建成一批特色鲜明、在国内同类教育中具有带头作用的示范专业,推动高职高专教育的改革与发展。

教育部《计划》和专业试点等新世纪高职高专教改项目工作开展以来,各有关高职高专院校投入了大量的人力、物力和财力,在高职高专教育人才培养目标、人才培养模式以及专业设置、课程改革等方面做了大量的研究、探索和实践,取得了不少成果。为使这些教改项目成果能够得以固化并更好地推广,从而总体上提高高职高专教育人才培养的质量,我们组织了有关高职高专院校进行了多次研讨,并从中遴选出了一些较为成熟的成果,组织编写了一批"新世纪高职高专教改项目成果"教材。这些教材结合教改项目成果,反映了最新的教学改革方向,很值得广大高职高专院校借鉴。

新世纪高职高专教改项目成果教材适用于高等职业学校、高等专科学校、成人高校及本科院校举办的二级职业技术学院、继续教育学院和民办高校使用。

高等教育出版社

2002年11月30日

前　言

从计算机技术与各项科学技术比较来看，计算机技术无疑是当代发展最为迅猛的科学技术之一。无论是文化领域、科技领域、金融领域还是军事领域，计算机技术的应用非常普遍，计算机技术已渗透到生产、生活的各个方面。

随着对计算机需求和依赖的与日俱增，计算机系统的规模和复杂性急剧增加，其软件开发成本以及由于软件故障而造成的经济损失也不断增加，软件质量问题已成为人们共同关注的焦点。

软件开发商要想最大限度地占有市场，必须把软件质量作为企业始终追求的重要目标之一，这样才能在激烈的竞争中不被淘汰出局。用户为了保证自己业务的顺利完成，当然也希望选用优质的软件。在一些关键应用中，如民航订票系统、银行结算系统、证券交易系统、自动飞行控制软件、军事防御和核电站安全控制系统等，都对软件质量提出了更高的要求。使用质量欠佳的软件，很可能造成灾难性的后果，如美国爱国者导弹防御系统、欧洲阿丽亚娜五型火箭发射失败问题、美国航空总署火星探测器坠毁灾难、千年虫问题、Intel 芯片浮点除法软件故障等，都是因使用质量欠佳的软件而造成的。因此，许多科学家在展望 21 世纪计算机科学发展方向和策略时，都把软件质量放在优先于提高软件功能和性能的地位。

软件测试是对软件需求分析、设计规格说明和编码的最终复审，是软件质量保证的关键步骤，是为了发现故障而执行程序的过程。随着软件系统规模和复杂性的增加，进行专业化、高效软件测试的要求越来越高，软件测试职业的价值越发显著，软件评测中心如雨后春笋般迅猛发展起来。可以预测，在未来 3 ~ 5 年内，软件测试技术将作为一门新兴产业而快速发展起来。

十分遗憾的是，在国内大量的出版物中，有关软件测试技术的书籍少之又少。近年来，我们在软件测试技术方面开展了一些调查、研究、分析和实践活动，对在软件测试当中可能遇到的一些关键问题进行了理论探讨并积累了一些宝贵的资料。在此基础上，着手编写本书，旨在介绍软件测试的基本概念、常用方法和技术，为普及我国的软件测试技术尽自己一份绵薄之力。同时希望读者能够以此为起点，学会如何选择有效的测试方法，学会如何迅速地找出软件中存在的故障，学会如何清楚地报告发现的软件问题，掌握软件测试的基本技术并能应用到具体实践中。

在本书的编写过程中，王雪莲同学做了部分翻译和辅助工作，科学院计算所闵应骅研究员在百忙中抽出时间审阅了全书，在此一并表示衷心的感谢。

由于本书编写任务紧迫，加之作者水平有限，书中错误和不妥之处在所难免，真诚希望读者批评指正，提出意见和建议。

作者联系方式：lzhao@ mail. buct. edu. cn

编　者

2003 年 9 月于北京

目　录

第1章 绪 论

从计算机诞生至今,计算机技术无疑成为当代发展最为迅猛的科学技术之一。今天,计算机已渗透到人们生活的各个方面。随着软件系统规模和复杂性的增加,软件开发成本以及由于软件故障而造成的经济损失也正在增加,软件质量问题已成为人们共同关注的焦点。软件测试技术也因此得到了飞速的发展,具体表现在新的软件测试方法、测试手段层出不穷,软件测试自动化程度不断提高,软件评测中心如雨后春笋般成长起来。可以预测,在未来3—5年内,软件测试技术将作为一门新兴产业而快速发展起来。

软件测试是在软件投入运行前,对软件需求分析、设计规格说明和编码的终审,是发现软件故障,保证软件质量,提高软件可靠性的主要手段。随着人们对软件质量的重视程度越来越高,软件测试在软件开发中的地位也将越来越重要。

本章重点:

- 软件测试
- 软件测试与软件可靠性
- 软件测试的发展历史

1.1 计算机系统中软件的可靠性问题

随着对计算机需求和依赖的与日俱增,计算机系统的规模和复杂性急剧增加,使得计算机软件的数量以惊人的速度急剧膨胀。例如,航天飞机机载系统有近500 000行代码,地面控制和处理系统大约有350 000行代码。美国电信业中,电信线路的正常运转需要数百个软件系统的支持,其代码总量超过一亿行。与此同时,计算机出现故障引起系统失效的可能性也逐渐增加。由于计算机硬件技术的进步,元器件可靠性的提高,硬件设计和验证技术的成熟,硬件故障相对显得次要了。研究表明:由于软件设计故障引起的系统失效与由于硬件设计故障引起的系统失效比是10:1。表1-1给出了计算机系统失效源的统计数据。

表1-1 计算机系统失效源的统计数据

系 统	数据发表年份	硬件(%)	软件(%)	维护(%)	操作(%)	环境(%)
AT&T ESS	1978	20	15	—	65	—
Tandem	1985	18	26	25	17	14
Bellcore	1986	26	30	—	44	—
Tandem	1987	19	43	13	13	12

由表1-1可以看出:软件故障正逐渐成为导致计算机系统失效和停机的主要因素。

由于人们对复杂计算机系统需求的急剧增加远远超过计算机软硬件设计、实现、测试及维护的能力,从而出现了许多可怕的计算机工程事故,其中大多数都是由于软件故障所致。1996 年 6 月 5 日,欧洲阿丽亚娜 5 型火箭第一次发射,由于定位软件出错,使计算机命令固态推进器与主发动机尾喷管发生偏离,导致火箭发射升空 40 秒后爆炸。1992 年 10 月 26 日,伦敦救护中心的计算机辅助发送系统刚刚启动就崩溃了,导致这个全世界最大的每天要接运五千多病人的救护服务机构全部瘫痪。1991 年美国爱国者导弹防御系统首次应用在海湾战争对抗伊拉克飞毛腿导弹的防御战争中,尽管该系统的赞誉不绝于耳,但是它确实在几次对抗导弹战役中失利,其中一枚在沙特阿拉伯的多哈击毙 28 名美国士兵。分析专家发现症结在于一个软件故障,一个很小的系统时钟错误积累起来就可能延迟 14 个小时,造成跟踪系统准确度下降。在多哈袭击战中,这样一个小故障造成系统被拖延 100 多个小时。1990 年 1 月 15 日,某通信中转系统软件发生故障,导致主干远程网大规模崩溃,使数以千计的电信运营公司损失惨重。1983 年,美国科罗拉多河河水泛滥,由于计算机对天气形势预测有误,水库未能及时泄洪,造成严重的经济损失和人员伤亡。1979 年,新西兰航空公司的一架客机因计算机控制的自动飞行系统发生故障而撞在阿尔卑斯山上,机上 257 名乘客全部遇难。千年虫问题是一个众所周知的软件故障,大约 1974 年,一位负责公司工资系统的程序员,由于当时使用的计算机存储空间很小,他尽量节省每一个字节,其中一个方法就是把 4 位数日期(例如 1973)缩减为 2 位数(73)。因为工资系统极其依赖数据处理,他节省了可观的存储空间。尽管他知道在计算 00 或 01 这样的年份时会出现问题,但他可能认为在 25 年之内程序肯定会更改或升级,而且眼前的任务比计划遥不可及的未来更加重要。到 1995 年,他编制的程序仍然在使用,而他却退休了。为此,全世界付出了数亿美元的代价来更换或升级类似的程序以解决千年虫问题。类似的例子,国内也可举出很多,大大小小的软件故障几乎每天甚至每时每刻都在发生,只不过有些问题不那么严重,人们没有注意到罢了。据不完全统计,由于软件故障,全世界已有 4 000 多人丢掉了生命。

随着信息技术的飞速发展,软件产品已应用到社会的各个领域,软件质量问题已成为人们共同关注的焦点。软件开发商为了占领市场,必须把软件质量作为企业的重要目标之一,以免在激烈的竞争中被淘汰出局。用户为了保证自己的业务顺利开展,当然希望选用优质的软件,质量欠佳的软件产品不仅会使开发商的维护费用和用户的使用成本大幅增加,还可能产生其他的责任风险,造成公司信誉下降。在一些关键领域的应用系统,如民航订票系统、银行结算系统、证券交易系统、自动飞行控制软件、军事防御和核电站安全控制系统中,对软件质量提出了更高的要求。使用质量欠佳的软件,还可能造成灾难性的后果。

因此,许多科学家在展望 21 世纪计算机科学发展方向和策略时,把软件质量放在优先于提高软件功能和性能的地位。但是,人们开发优质软件的能力大大落后于社会对计算机软件不断增长的需求,开发出的软件系统普遍存在许多隐藏的故障和缺陷。毋庸置疑,提高软件质量,如同提高软件生产率一样,已成为整个软件开发过程中必须始终关心和设法解决的问题。

1.2 软件测试与软件可靠性

人们对计算机依赖的程度越高,对其可靠性的要求就越高。从实验系统所获得的统计数据表明,运行软件的驻留故障密度,对于关键的财产软件为每千行代码 1 ~ 10 个故障,对于关键的

生命软件为每千行代码 0.01 ~ 1 个故障。然而,正是由于软件可靠性的大幅度提高才使得计算机广泛应用于社会的各个方面。一个可靠的软件应该是正确的、完整的、一致的和健壮的。IEEE(电气与电子工程师学会)将软件可靠性定义为:系统在特定的环境下,在给定的时间内,无故障地运行的概率。用来评价软件按照用户的要求和设计目标,完成规定功能的能力,涉及软件的性能、功能、可用性、可服务性、可安装性、可维护性以及文档等多方面特性,因此,软件可靠性是对软件在设计、开发以及在它所预定环境中具有能力的置信度的一个测度,是衡量软件质量的主要参数之一。

关于软件可靠性方面的量度,主要依据以下几点:

- 软件中初始故障的个数。
- 软件经过测试后,通过查错、改错,在软件中剩余的故障个数。
- 平均无故障时间。
- 故障间隔的时间长度。
- 故障发生率。
- 经预测,下一次故障的发生时间等。

对软件系统中可能出现的故障进行分类,有利于软件可靠性分析工作的进行。故障一般可分为:硬件故障、软件故障、操作故障和环境故障。硬件故障是由物理性能的恶化造成的;软件故障是由设计阶段的人为因素造成的;操作故障是指操作人员和维护人员的错误;环境故障则包括电源、外界干扰、地震、火灾、病毒等各种外界因素引起的故障。故障可以形式化地定义为软件在其执行期间的表现偏离了事先规定的行为要求。如果规格说明书错了,尽管软件的实现与规格说明的要求相符,但它与用户的要求不吻合,从用户立场上来看,这也是对事先规定行为的偏离,它将直接影响到用户的使用。因此,只要用户有抱怨,就可以说,软件出现了故障。

实际上,对于软件来讲,不论采用什么样的技术和方法,软件中都会有故障存在。采用新的编程语言、先进的开发方式、完善的开发过程,可以减少故障的引入,但是不可能完全杜绝软件中故障的存在,这些软件故障需要靠测试来发现,软件中的故障密度也需要靠测试来估计。软件测试是对软件需求分析、设计规格说明和编码的终审,是软件质量保证的关键步骤。如果给软件测试下定义,可以这样讲:软件测试是为了发现故障而执行程序的过程。或者说,软件测试是根据软件开发各阶段的规格说明和程序的内部结构而精心设计的一批测试用例,并利用这些测试用例去执行程序,以发现软件故障的过程,其根本目的是以尽可能少的时间和人力发现并改正软件中潜在的各种故障及缺陷。实际上,测试工作一直对准软件中隐含的各种故障,所有的测试方法和手段都是以找出软件中隐含的故障为目的的。软件中隐藏的故障数目,直接决定软件的可靠性。如果不能将软件中隐含的故障及时排除,一旦暴露出来就会给使用者和维护者带来不同程度的严重后果。所以,软件测试必须在软件投入生产运行之前进行,以尽可能多地发现软件中存在的故障,提高软件可靠性。

软件可靠性模型利用软件测试所提供的有关软件系统的故障数据,估算软件的可靠性,对软件将来的故障行为进行预测,以协助开发人员监督软件开发过程,辅助软件过程管理,如过程评估、风险分析、项目估计与决策等。因此,软件测试是保证软件质量,提高软件可靠性的主要手段。

随着人们对软件测试重要性认识的加深,软件测试在整个软件开发周期中所占的比例日益

增大。目前,许多软件开发机构已将研制力量的40%以上花费在软件测试中。特殊情况下,对于要求高可靠性的软件,例如飞行控制、核反应堆监控软件等,其软件测试费用甚至高达软件开发其他阶段所用费用总和的3~5倍。

1.3 软件测试的发展历史、现状和展望

从计算机问世以来,软件的编制与测试就同时摆在人们的面前。早在20世纪50年代,英国著名的计算机科学家图灵就给出了软件测试的原始定义。他认为,测试是程序正确性证明的一种极端实验形式。早期测试主要针对机器语言和汇编语言,设计特定的测试用例,运行被测程序,将所得结果与预期结果进行比较,从而判断程序的正确性。测试用例一般在随机选取的基础上,吸取测试者的经验或是凭直觉判断出某些重点测试区域。但测试在软件开发中的作用并没有受到应有的重视。测试方法和理论研究进展缓慢,除去一些非常关键的程序系统外,一般程序的测试大都是不完备的。在开发工作结束后,含有大大小小缺陷的程序投入运行了。这些隐藏的缺陷一旦暴露出来就会给用户和维护者带来不同程度的严重后果。早年火星探测运载火箭因控制程序中错写了一个逗号而爆炸,对空防御系统曾把月亮当做洲际导弹的目标来轰击,都已成为人们谈论测试时的笑柄。

测试工作未受到重视的另一个原因是人们的心理因素。从软件系统开发者的角度,研制工作的目标是使系统能够运转起来,这是富有刺激性和创造性的任务,当付出的精力逐渐变为成果时,人们不愿做那些后续的既麻烦又可能否定自己成果的测试工作,也不愿意让别人给自己开发的软件挑毛病。正如Myers所说的那样,软件测试是设法从程序中找错的破坏性过程。测试人员和开发人员的这一对抗心理,在一段时间内成为测试工作的障碍,极大地影响了测试技术的发展。

直到20世纪70年代以后,测试的意义才逐渐被人们认识,软件测试的研究才开始受到重视。F. P. Brooks总结了开发IBM OS/360操作系统中的经验,阐明了软件测试在大型系统研制中的重要意义。1975年,Goodenough首次提出了软件测试理论,从而把软件测试这一实践性很强的学科提高到了理论的高度。随后,Huang全面地讨论了测试准则、测试过程及测试数据生成等软件测试问题。W. C. Hetzel整理出版了*Program Test Methods*一书,总结归纳了测试方法以及各种自动测试工具,这是软件测试的第一本著作。随后E. P. Miller在测试管理和普及方面做了大量工作,为把现代测试概念推向实践做出了重要贡献。1982年,美国北卡罗莱纳州大学召开了首次软件测试技术会议,这是软件测试与软件质量研究人员和开发人员的第一次聚会,这次会议成为软件测试技术发展的一个重要的里程碑。此后,测试理论、测试方法进一步完善,从而使软件测试这一实践性很强的学科成为有理论指导的学科。

在软件测试理论迅速发展的同时,各种高级的软件测试方法也将软件测试技术提高到了无法比拟的高度。J. C. Huang提出了程序插装的概念,使被测程序在保持原有逻辑完整性的基础上,插入“探测仪”,以便获取程序的控制流及数据流信息,并可得到测试的覆盖率。W. E. Howden对路径测试进行了深入的研究,提出了系统功能测试及代数测试等概念。L. A. Clarke等人将符号执行的概念引入到软件测试中,提出了符号测试方法,并且建立了DISSET等符号测试系统。R. A. Demillo提出了基于程序变异的测试方法,使传统的测试技术领域增加了新的成

员——错误驱动测试。1977 年 L. Osterweit 等人首先引入了数据流测试方法,通过对数据流进行静态分析以找出程序中潜藏的缺陷。1983 年 Ryoichi Hosoya 等在数据流测试方法中加入了变量值域分析,使数据流方法能检测出更多类型的软件缺陷。1988 年 Frankl 将数据流信息应用到路径选择中,并定义了相应的测试覆盖准则,如所有定义,引用路径覆盖准则、所有定义覆盖准则、所有计算引用覆盖准则等,并给出了各种覆盖准则之间的包含关系,如 all-du-paths ⊂ all-uses 等。

1980 年针对程序域错误,L. White 等提出了一种新的测试策略——域测试策略,以后发展成为一个有效的模块测试方法。1990 年 Korel 对给定路径上不满足要求的分支,利用分支函数极小化,实现了测试数据的自动生成。1991 年 DeMillo 提出了一种基于故障的测试数据生成方法,用代数约束来描述检测特定类型故障的测试数据。1998 年 Gupta 利用迭代逼近法,求取满足给定路径上所有谓词的测试数据。Gotlieb 提出用约束求解的方法寻找经过给定语句的路径,生成相应的测试数据。

20 世纪 70 年代,软件工程的概念逐渐形成。把软件工程活动分为需求分析、设计、编码、测试和维护几个阶段的软件生存期的概念被人们广泛接受。同时,在软件开发的实践中,人们也逐渐认识到,在开发初期发现并解决软件故障所付出的代价远比在编码以后经过测试而发现故障并加以改正的代价小得多。各种有关生存期前几个阶段的测试理论和测试方法应运而生。W. E. Howden 给出了软件生存期各个阶段的测试目标。1986 年 L. J. Hayes 提出规格说明指导的模块测试方法,主张在早期开发中就考虑测试的需求。1988 年 Hall 提出了一种利用 Z 规格说明进行软件测试数据生成的方法。1990 年 Tsai 提出从关系代数查询表示的规格说明中生成测试数据的方法。1994 年 Weyuker 对基于过程控制的形式化规格说明方法进行改进,提出了基于布尔规格说明的测试数据生成方法。1995 年 Marick 给出了许多软件测试工程方面的技巧,包括从规范说明寻找测试线索,借助测试需求目录,产生测试需求清单,通过指派确切的输入和期望的输出值,形成测试规范,进而测试程序的全过程。

随着面向对象分析和面向对象设计技术的日渐成熟,面向对象的软件开发技术得到了软件界的普遍认可,面向对象软件测试技术的研究逐渐受到人们的重视。1994 年 9 月 Communication of ACM 出版了面向对象的软件测试专集,涉及了类测试、集成测试和面向对象软件的可测试性等问题。1995—1997 年的 Object 杂志,由 R. Binder 执笔的面向对象的软件测试专栏,重点讨论了基于状态的面向对象测试技术。

面向对象的程序中,对象是封装了描述其属性的数据以及可以施加在这些数据上的操作的封装体。属性表示对象的状态,操作表示对象的行为,消息则描述了对象执行操作的规格说明。对象之间通过发送消息启动相应的操作,通过修改对象的状态,实现系统状态间的相互转换。类是对具有相同属性和行为的一组相似对象的描述,它描述了该类对象所具有的共同特征。1989 年 Fiedler 从面向对象的测试与传统测试的不同点出发,提出了面向对象单元测试的解决方案,从此开始了面向对象软件测试的研究工作。随后,Harrold 较系统地考虑了类间的继承测试,提出了一种根据类间继承关系的层次特性对类进行增量测试的技术,其特点是通过复用和增量更新父类的测试信息去指导子类的测试。Doong 根据类是抽象数据类型的实现这一原理,引入了一种与面向对象语言语法相似的代数规范描述语言 LOBAS,作为类的测试模型。我国著名软件测试专家陈火炎教授提出了一种基于代数规范描述的黑白盒集成类测试方法,该方法用黑盒测试选取测试用例,用白盒测试来检测程序执行一个测试用例时产生的两个对象是否处于相同的

抽象状态,并补充一些测试用例,对类进行测试。

近年来,尽管软件测试技术与实践有了很大的进展,但总的来说,仍然与软件开发实践的要求相距较远。就目前软件工程发展的状况而言,软件测试仍然是较为薄弱的一个方面。不仅测试理论,已有的测试方法也不能满足当前软件开发的实际需求。为此,国际上每两年召开一次软件测试与分析研讨会,专门就软件测试与软件质量问题进行广泛的交流。为推进和协调软件测试的研究工作,1999 年在美国洛杉矶召开的第 21 届国际软件工程会议上,将软件测试作为一个技术专题展开,以改善软件开发过程,提高软件质量。我国每两年召开一次的全国软件工程会议、全国容错计算会议都设有软件测试专题部分。2001 年首次召开的全国测试学术会议,将软件测试作为一个主要的议题,以推进我国软件测试工作的研究。

为了提高软件测试效率,加快软件开发过程,一些测试工具相继问世,如静态分析工具、测试数据生成工具、测试评估工具以及将多种测试工具融为一体的集成化测试系统等。

(1) 静态分析工具

静态分析工具是在不执行程序的情况下,分析软件的特性。静态分析主要集中在需求文档、设计文档以及程序结构上,可以进行类型分析、接口分析、输入输出规格说明分析等。常用的静态分析工具有:ViewLog 公司开发的 LogiScope 分析工具;Software Research 公司开发的 TestWork/Advisor 分析工具及 Software Emancipation 公司开发的 Discover 分析工具等。

(2) 测试数据生成工具

测试数据生成工具可以为被测程序自动生成测试数据,减轻人们在生成大量测试数据时付出的劳动,同时还可避免测试人员对一部分测试数据的偏见。常用的测试数据生成工具有:Bender & Associates 公司提供的功能测试数据生成工具 SoftTest;International Software Automation 公司提供的 Panorama C/C ++ 测试数据生成工具;Parasoft 公司提供的 C/C ++ 单元测试工具 Parasoft C ++ test 及 Java™ 类测试工具 Parasoft jtest 等。

(3) 测试评估工具

测试评估工具用来评估程序结构元素被覆盖的程度,从而确定测试运行的充分性。常见的测试评估工具有:Bell 中心开发的 C 程序测试覆盖分析工具 ATAC;Rational 公司开发的 PureCoverage;Software Research 公司开发的 TestWorks/Coverage 及 Parasoft 公司开发的 Parasoft test 测试覆盖率分析工具 CMC ++ 等。

(4) 集成化测试系统

集成化测试系统将多种测试工具融为一体,是一种功能较强的测试工具。常见的有:Microsoft 公司开发的对 Windows 应用程序进行自动测试的集成化测试系统 Microsoft Test for Windows;美国 Parasoft 公司开发的自动故障检测系统 Parasoft insure ++ 以及 PureArtia 公司推出的一组在多平台上测试 C/C ++ 和 FORTRAN 程序代码的测试工具,以帮助 Windows 和 UNIX 平台上使用 C/C ++ 或 FORTRAN 语言的用户解决软件集成的质量问题。

软件测试在我国起步较晚,最初主要在项目组内部进行手工测试。近年来,随着计算机在我国许多重要部门的广泛使用,软件测试的研究逐渐被重视起来,并取得了一些初步的研究成果。例如,中国科学院计算技术研究所进行的软件测试方法研究,北京大学计算机科学系进行的面向对象软件测试工具、测试建模和测试数据生成技术及相关问题的研究;北京航空航天大学开发的 C 软件分析与测试工具 SafePro/C 及 C ++ 软件分析与测试工具 SafePro/C ++;南京大学软件

新技术国家重点实验室则着重研究软件复杂度分析和动态测试问题,开发了计算机辅助面向对象软件测试工具包 OOAK ;以及中国空间技术研究所、北京控制工程研究所进行的星载计算机软件测试环境及实现的研究等。

小结

研究表明 :由于软件设计故障引起的系统失效与由于硬件设计故障引起的系统失效比是 10: 1。软件故障正逐渐成为导致计算机系统失效和停机的主要因素。

不论采用什么样的技术和方法,软件中都会有故障存在。只要用户有抱怨,就可以说,软件已出现了故障。

软件可靠性是对软件在设计、生产以及在它所预定环境中具有能力的置信度的一个测度,是衡量软件质量的主要参数之一。而软件测试是对软件需求分析、设计规格说明和编码的最终复审,是软件质量保证的关键步骤。软件测试必须在软件投入生产运行之前进行,以尽可能多地发现软件中的故障,提高软件可靠性。

第 1 章习题

1. 简述软件测试与软件可靠性的关系。
2. 衡量软件质量的主要参数有哪些?
3. 有没有质量很高但可靠性很差的产品,试举例说明。

第 2 章　软件测试的实质

随着人们对软件质量的重视程度越来越高,软件测试在软件开发中的地位越来越重要。软件测试是目前用来检验软件能否完成预期的功能的惟一有效的方法,其总目标是充分利用有限的人力和物力资源,高效率、高质量地进行测试。

本章重点:

- 软件测试的基本概念
- 软件测试的关键问题
- 软件测试的复杂性与经济性
- 软件测试原则
- 停止测试的标准

2.1　软件测试的基本概念

2.1.1　软件测试的目的

软件危机曾经是软件界甚至整个计算机界最热门的话题之一。为了解决这场危机,软件从业人员、专家和学者做出了大量的努力。现在人们已经逐步认识到软件危机实际上是一种状况,那就是软件中存在故障,正是这些故障导致了软件开发在成本、进度和质量上的失控。由于人的主观认识常常难以完全符合客观现实,与工程密切相关的各类人员之间的通信和配合也不可能完美无缺。因此对于软件来讲,不论采取什么样的技术和方法,软件中都会有故障存在,即使标准商业软件中也有故障存在,只是严重程度不同而已。采用新的编程语言、先进的开发方式、完善的开发过程,可以减少故障的引入,但是无法完全杜绝软件中的故障。

测试在软件开发过程中一直备受关注,即使在传统的软件工程中,也有一个明确、独立的测试阶段。随着软件危机的频频出现以及人们对于软件本质的进一步认识,软件测试的地位得到了前所未有的提高。尽管软件测试的技术性很强,但正确对待软件测试对测试的成功具有深远的影响。然而,究竟什么是软件测试,长期以来一直存在着不同的观点。目前仍然有许多人错误地看待软件测试,这也正是不能很好地做好软件测试工作的原因之一。因为人类的活动具有高度的目的性,建立适当的目标具有重要的心理作用,因此,正确认识测试的目的十分重要。测试目的决定了测试方案的设计,如果测试的目的是要证明程序中没有隐藏的故障存在,那就会不自觉地回避可能出现故障的地方,设计出一些不易暴露故障的测试方案,从而使程序的可靠性受到极大的影响。相反,如果测试的目标是要证明程序中有故障存在,那就会力求设计出最能暴露故障的测试方案。软件测试是一项花费昂贵的活动,测试者希望通过测试来提高软件的质量或可靠性,这就意味着要发现并改正程序中的错误。所以,进行测试时不应该为了显示程序是好的,而应该从软件中含有故障这个假定出发去测试程序,从中发现尽可能多的软件故障。因此,像

“软件测试是证明程序中不存在故障的过程”,“测试的目的是要提供有说服力的证据证明软件没有故障,或是显示某种特殊类型的故障不存在”等,都是软件测试的错误定义。

1. 不同时期关于软件测试的定义

① 为了发现故障而执行程序的过程。

② 确信程序做了它应该做的事。

③ 确认程序正确实现了所要求的功能。

④ 以评价程序或系统的属性、能力为目的的活动。

⑤ 对软件质量的度量。

⑥ 验证系统满足需求,或确定实际结果与预期结果之间的差别。

定义①强调寻找故障是测试的目的,定义②、③侧重于用户满意程度,定义④、⑤强调评估软件质量,而定义⑥则将重点放在预期结果上。这些定义虽说法不一,但都很有用。这里没必要争论它们之间的异同,但了解各种不同的定义,有助于集中精力解决那些测试中真正需要解决的问题。

1983 年 IEEE(国际电子电气工程师协会)提出的软件工程标准术语中给软件测试下的定义是:

使用人工或自动手段来运行或测定某个系统的过程,其目的在于检验它是否满足规定的需求或是弄清预期结果与实际结果之间的差别。

该定义包含了两方面的含义:

- 是否满足规定的需求。
- 是否有差别。

如果有差别,说明设计或实现中存在故障,自然不满足规定的需求。因此,这一定义非常明确地提出了软件测试以检验软件是否满足需求为目标。该定义指出测试时需要明确给定预期结果,然后将它们与实际结果进行比较,因此比较是测试的基础之一,确定预期输出是测试用例必不可少的一部分。如果事先无法确定预期的测试结果,由于“人们常看见他想看见的东西”,往往会把看起来似是而非的东西当成正确的结果。换句话说,人们会下意识地盼望看到正确的结果。解决这个问题的一种方法是,提倡用事先精确写出的程序执行的预期输出结果来检查所有的输出。因此,一个测试用例由两部分组成:对程序输入数据的描述和由这些输入数据应产生的预期结果的精确描述。

心理学研究告诉人们,当一个人在做一件知道是不合适的或不可能做到的事情时,往往会做不好。例如,如果让一个人在 15 分钟内解出一个刊登在《纽约时报》周末版上的交叉填字字谜,那么,10 分钟后也许他没有任何进展,因为他可能因认为自己做不到而放弃努力。然而,如果要求花 4 个小时来求解这个问题,也许他在最初的 10 分钟内就有了较大的进展。Myers 把软件测试定义为在程序中找出故障的过程,使测试成为可以做到的任务,从而克服了心理上存在的问题。因此,针对软件测试人员而言,测试的最好定义是:

测试以发现故障为目的,是为了发现故障而执行程序的过程。

强调正确定义软件测试的另一方面是正确使用“成功”和“失败”这两个词。大多数工程管理人员把没有发现故障的测试称为“成功”的测试,而把查出故障的测试称之为“失败”的测试。一般来说,“成功”表示达到了某种目的,而“失败”则表示令人失望或事不如愿。然而,在测试

中,由于查不出故障的测试浪费了大量的时间和人力,因此使用“成功”这个词就显得不够恰当了。同样,也不能把查出故障的测试看成是“失败”的测试。恰恰相反,事实证明,发现故障是一个有价值的工作。考虑一个类似的情况,一个人感到全身不舒服去看病。如果医生做了一番检查后,却诊断不出他得了什么病,那就不能说这些检查是“成功”的。相反,这些检查应该看作是失败的。因为病人白白花了75元的检查费,却什么也没有被诊断出来。如果检查确定病人患有胃溃疡,那检查就是成功的。这时医生就可以对症治疗了。显然,当测试一个程序时,可以把它当做一个病人。所以,强调测试正确定义的另一办法,是把这两个词的习惯用法颠倒过来。将发现错误的测试称为成功的测试,而使程序产生正确结果的测试称为失败的测试,那么,“一个好的测试就是有可能发现至今尚未被发现的故障的测试”;“一个成功的测试是发现了至今未被发现的故障的测试”。

2. 关于软件测试的一些常用术语

(1) 测试

① 测试是一种活动,在该活动中一个系统或组成部分在特定条件下被运行,结果被观察或记录,并对该系统或组成部分的某些方面进行评估。虽然测试有两个显著的目标,找出故障或演示软件执行正确。

② 测试是一个或多个测试用例的集合。

(2) 测试用例

① 测试用例是为特定的目的而开发的一组测试输入、执行条件和预期结果。

② 测试用例是执行的最小实体。

(3) 测试步骤

测试步骤详细说明了如何设置、执行和评估特定的测试用例。

图2-1给出了一个测试生命周期模型。从图2-1可以看出,有3个阶段(①、②、③)可能引入故障,或导致产生通过其他阶段的故障。一位测试专家将测试生命周期归纳为:前3个阶段“引入故障”,测试阶段“发现故障”,而后3个阶段(⑤、⑥、⑦)则“清除故障”。第7个阶段“故障清除”有可能导致以前正确执行的软件出现了错误的行为,引入另一个新的故障。

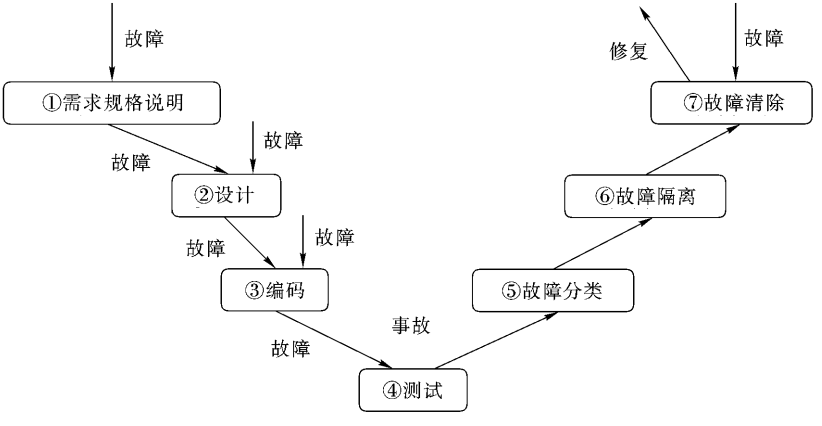


图2-1 一个测试生命周期

2.1.2 软件测试涉及的关键问题

软件测试主要涉及以下 5 方面的问题。

(1) 谁来执行测试

一个软件产品的开发通常涉及开发者和测试者两种角色。开发者通过开发而形成产品,例如分析、设计、编码、调试或者文档编制等。测试者则通过测试来检测产品中是否存在缺陷,包括根据特定的目的设计测试用例、构造测试、执行测试以及评估测试结果等。一般的做法是:开发机构负责他们自己代码的单元测试,而系统测试则由一些独立的测试人员或专门的测试机构进行。

(2) 测试什么

很显然,表现在程序中的故障,并不一定是编码所引起的。很可能是详细设计、概要设计阶段,甚至是需求分析阶段的问题引起的。即使针对源程序进行测试,所发现故障的根源也可能在开发前期的各个阶段。解决问题、排除故障也必须追溯到前期的工作。实际上,软件需求分析、设计和实施阶段是软件故障的主要来源,因此,需求分析、概要设计、详细设计以及程序编码等各个阶段所得到的文档,包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序,都应成为软件测试的对象。

当因进度、人力、物力等原因,不可能对软件的每一部分进行完全测试时,应采取什么策略来设计测试用例呢?是随机生成测试用例?还是测试软件中常用的功能或高风险的部分?不言而喻,将精力花在系统的常用功能或高风险的部分是适当的。

(3) 什么时候测试

测试是一个与开发相并行的过程,还是当开发取得一定阶段性成果之后的活动或是开发结束之后的活动?也就是说,模块开发结束之后可以进行测试,也可以推迟至各模块装配成一个完整的程序之后再进行测试。实践表明,随着开发的深入,未进行测试的模块对整个软件的潜在破坏作用就越明显。

那么应该在什么时候进行测试才是恰当的呢?有时测试只需要在开发过程即将结束时进行,也就是说,系统测试或验收测试是对软件的惟一正式的测试。在开发者的数量相对少的时候,这种方法还可行。但对大多数开发过程来说是不适合的。人们已经开始认识到:测试开始得时间越早,测试执行得越频繁,所带来的整个软件开发成本的下降就会越多。测试的另一个极端是每天都进行测试,一旦软件的模块开发出来就对它们测试,这样显然又会延缓早期开发的进度。不过,它能够大大减少将所有模块装配到项目中以后出现问题的可能性。

(4) 怎样进行测试

软件“规范”说明了软件本身应该达到的目标,程序“实现”则是一种对应各种输入如何产生输出结果的算法。简言之,规范说明一个软件要做什么,而程序实现则规定了软件应该怎样做。对软件进行测试就是根据软件的功能规范说明和程序实现,利用后续各章介绍的各种测试方法,生成有效的测试用例,对软件进行测试。

(5) 测试停止的标准是什么

从现实和经济的角度来看,对软件进行完全测试是不可能的。那么,什么时候停止测试呢?因为无法判断当前查出的故障是否为最后一个故障,所以决定什么时候停止测试是一件很困难

的事。测试完成的传统标准是分配的测试时间用完了或完成了所有的测试又没有检测出故障。但这两个完成标准都没有什么实用价值。

实用的停止测试标准应该基于以下几个因素：

- 成功地采用了具体的测试用例设计方法。
- 每一类覆盖的覆盖率。
- 故障检测率(即每一单元测试时间内检测出的故障数)低于指定的限度。基于故障检测数量的标准必须注明故障的严重性程度。
- 检测出故障的具体数量(估计存在故障总量的比率)或消耗的具体时间等。

2.1.3 软件测试与软件质量保证

故障在软件开发和维护的任何阶段都可能产生,可能还会造成时间、财产、客户甚至生命的流失。测试能帮助确保一个软件产品满足需求,但测试并不是质量保证。有些人错误地将软件测试和软件质量保证等同起来。在许多组织中,软件质量保证部门通常负责开发测试计划和执行系统测试,也可能会对开发过程中的测试进行监测和保留统计数据。软件测试是任何软件质量保证过程中必需的但并不是所有的部分。软件质量保证部门从事的是那些用来防止和去除软件缺陷的活动,负责制订为了生产出更好的软件而应该遵守的标准,包括定义为理解设计意图而创建的各种文档的类型,指导项目活动的过程以及量化决议结果的方法等。

通过测试可以发现软件故障,对一个系统做的测试越多,就越能确保它的正确性。不言而喻,大量的软件测试将提高软件的质量。然而,软件测试通常不能保证系统的运转 100% 正确。因此,软件测试在确保质量方面的主要贡献在于它能发现那些在一开始就应该能够避免的错误。软件质量保证的使命首先是避免错误。要做到这一点,除了测试外还需要其他方面的处理。

2.2 软件故障

2.2.1 故障定义

故障(fault)、失效(failure)、错误(error)、缺陷(defect)、隐错(bug)、过失(mistake)、异常(anomaly),这些术语常用来描述软件失效时的现象。之所以有这么多含义相近的术语来描述软件故障是由于软件开发公司的文化和公司用于开发软件的过程不同所造成的。本书采用 IEEE 制定的标准术语。

- 错误(error)。人是会犯错误的。一个很接近的同义词是过失(mistake)。过失是人犯下的,是人做的一件错事或人为产生的一个不正确结果。
- 故障(fault)。故障是错误的结果(可能导致失效)。更精确地说,故障是错误的表现。与故障很接近的一个同义词是缺陷(defect)。
- 失效(failure)。故障(例如崩溃)引起的结果(表现)。

过失是人犯下的,是人做的一件错事。人们在编写程序时会出错,比如,一时马虎按错了键,这些过失都有可能将一个故障或隐错(bug)带进产品中。

失效是故障的表现形式,即使故障或隐错没有引起失效,它们也是客观存在于文档或代码中

的,测试人员的任务就是找出它们。

当出现了失效,比如,系统崩溃、用户得到了一个错误的信息、取款机给出错误的钱数等,这些导致不正确结果的全体都是故障。

没有几个人能十分准确地使用这些术语,实际上也没有必要。本书主要使用故障(fault)和缺陷(defect)两个词。

2.2.2 软件故障分类

与任何事物一样,软件也有一个从孕育、诞生、成长到衰亡的生存过程,通常称为软件生存周期。包括制定计划、需求分析、设计、程序编码、测试及运行维护 6 个阶段。软件开发经过制定开发计划,进行需求分析、软件设计阶段之后,才能进入编写程序阶段,程序编写完之后还必须经过大量的测试工作才能交付使用。因此,编写程序只是软件开发过程的一个阶段。在典型的软件开发工程中,编写程序所需的工作量只是软件开发全部工作量的 20% 左右。分析软件故障分布情况,有助于将测试的主要精力更好地集中到最有价值的地方,以改进软件测试过程,提高软件测试的效率。

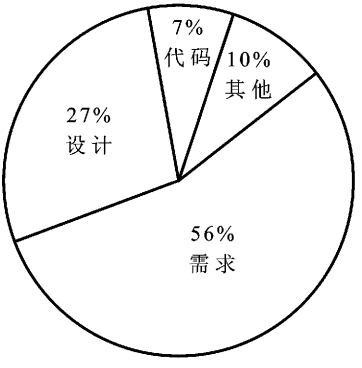


图 2-2 软件故障分布

软件故障有多种分类方法:可以以故障出现的开发阶段来划分,以失效产生的后果来划分,以解决难度来划分,以不解决可能会产生的风险来划分等。图 2-2 给出了一种以开发阶段来划分软件故障的大致分布情况。表 2-1 给出了一种根据故障后果严重程度来区分的分类方法。

表 2-1 按后果严重程度分类的软件故障

严重程度	举例说明
1(轻微)	拼写错误等
2(中等)	误导或重复信息等
3(使人不悦)	被截断的名称等
4(影响使用)	有些情况没有处理
5(严重)	丢失功能
6(较严重)	不正确的处理
7(很严重)	经常出现严重的错误
8(无法忍受)	数据库破坏
9(灾难性)	系统停机
10(传染性强)	影响其他系统停机

下面给出一些其他类型的软件故障,大部分摘自 IEEE 标准,也增加了一些普遍认为常见的软件故障。

(1) 软件需求故障

软件需求制定的不合理或不正确,需求不完整,需求分析文档有误,功能或性能的规定有误

等,例如,遗漏了某些功能或规定了某些冗余的功能,为用户提供的信息有误或信息不确切等。

(2) 输入/输出故障

输入故障主要表现在:不能接受正确的输入,接受了不正确的输入,参数有错或遗漏等。

输出故障主要表现在:输出格式有错,输出结果有错,在错误的时间产生正确的结果(太早、太迟),不一致或遗漏了结果,不合逻辑的结果,拼写/语法错误,修饰词错误等。

(3) 逻辑故障

属于逻辑故障的有:遗漏了情况、情况重复、边界条件出错、解释有误、遗漏条件、外部条件有错、不正确的循环迭代、错误的操作符等。

(4) 计算故障

不正确的算法、遗漏计算、不正确的操作数、不正确的操作、括号错误、精度不够(四舍五入、截断)以及错误的内置函数等都属于计算故障。

(5) 接口故障

接口故障包括:不正确的中断处理,I/O 时序有错,调用了错误的过程,调用了不存在的过程,参数不匹配(类型、个数)和不兼容的类型等。

(6) 数据故障

不正确的初始化,不正确的存储/访问,错误的标志/索引值,不正确的打包/拆包,使用了错误的变量,错误的数据引用,缩放数据范围或单位错误,不正确的数据维数,不正确的下标,不正确的类型、不正确的数据范围以及不一致的数据等都属于数据故障。

2.2.3 软件故障的修复费用

由于人的主观认识常常难以完全符合客观现实,与工程密切相关的各类人员之间的通信和配合也不可能完美无缺。所以,在软件生存周期的每个阶段都不可避免地会产生差错,并且前一阶段的故障自然会导致后一阶段相应的故障,因此故障会积累起来。此外,后一阶段的工作是前一阶段工作结果的进一步具体化,因此,前一阶段的一个故障可能会造成后一阶段中出现几个故障,也就是说,软件故障不仅有积累效应,还有放大效应。图 2-3 形象地表示了故障的积累和放大效应,这和制造业的装配线类似,如果一个坏零件或次品被允许上线,从这点开始,包含它的组件就是'坏'的,当它被装配到后面的组件时,往往会以新的形式出现。如果该组件下了线,并出了厂门,情况就会更糟糕,就得为那个坏零件付出更大的代价。

所有的错误都是要付出代价的。没有被发现的故障以及那些在开发过程中很晚才发现的故障都是修复成本最高的。没有被发现的故障将在系统中迁移、扩散,最终导致系统失效,造成严重的财产损失,有时还会带来法律上的麻烦,系统将终身为此付出高昂的代价。直到很晚才发现的故障常常造成代价昂贵的返工。Boehim 分析了 IBM、GTB、TRW 等一些软件公司的统计资料,发现在软件开发的不同阶段进行改动需要付出的代价完全不同。后期改动的代价比前期进行相应修改要高出 2~3 个数量级。图 2-4 显示了随着时间推移,软件故障的修复费用呈几何数级增加,也就是说,随着时间的推移,将会数十倍地增长。例如,在需求分析阶段早期发现软件故障,修复费用也许只要几角钱就够了。同样的软件故障如果到软件编制完成开始测试时才发现,修复费用可能就要花好几元钱。如果是用户发现的,修复费用可能就达上百元。

图 2-3 软件故障的积累和放大效应

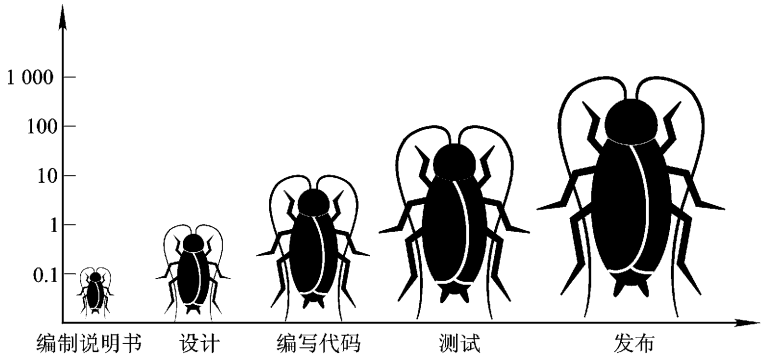


图 2-4 软件故障的修复费用

2.3 测试的复杂性与经济性

人们常常以为,开发一个软件是困难的,而测试一个软件则相对比较容易。初涉软件测试的人可能希望通过测试,找出所有软件故障。其实,要查出程序中所有的故障既不现实,也不可能。这一问题涉及软件测试的复杂性与经济性。

1. 测试复杂性

例如,要测试一个三角形程序,该程序完成下述功能:

输入3个整数a、b和c,作为三角形的3条边。通过程序判断由这3条边构成的三角形的类型是等边三角形、等腰三角形还是一般三角形并打印出相应的信息。

写下自己认为合适的测试输入,然后根据测试输入回答下面的问题,每回答1个"是"加1分,最后看看能得多少分。

① 是否设计了一种测试输入表示合法的一般三角形。

注意,像(1,2,3)和(2,3,9)这样的测试输入不应该回答"是",读者可以想想为什么。

② 是否设计了一种测试输入表示合法的等边三角形。

③ 是否设计了一种测试输入表示合法的等腰三角形。

注意,像(4,4,8)这样的测试输入不应该回答"是",因为不存在这样的三角形。

④ 是否至少设计了3种测试输入表示合法的等腰三角形,由此检查了两条边相等的所有3种排列方案?如(3,3,4)、(4,3,3)和(3,4,3)。

⑤ 是否设计了这样的测试输入,其中三角形的一条边长为0。

⑥ 是否设计了一种测试输入,其中3个整数都大于0,而其中的两数之和等于第三个数。

注意,如果把(2,3,5)当成一个一般三角形,则表明程序中有故障。

⑦ 是否至少设计了3种第⑥类那样的测试输入,检查一条边边长等于另外两边边长之和的3种排列方式?如(2,3,5)、(5,3,2)和(3,5,2)。

⑧ 是否设计了一种测试输入,表示3个整数都大于0,而其中某两个数的和小于第三个数?

注意,如果把(2,3,9)当成一个一般三角形,则表明程序中有故障。

⑨ 是否至少设计了3种第⑧类那样的测试输入,检查了一条边小于另外两边之和的3种排列方案。如(2,3,9),(2,9,3)和(9,2,3)。

⑩ 是否设计了一种测试输入,表示3条边边长都为0,即(0,0,0)。

⑪ 是否设计了这样的测试输入,其中三角形的一条边长为负数。

⑫ 是否至少设计了一种测试输入,其中三角形的边长不是整数。

⑬ 是否至少设计了一种测试输入,其中三角形的边数不是3。例如,给出两条边或四条边。

⑭ 对于每一种测试输入,是否还给出了预期的输出。

当然,满足上面条件的一组测试输入不能保证能检查出所有可能的故障,但由于问题①~⑬代表了该程序实际上可能发生的故障,对程序进行充分的测试应该能检查出这些故障。你得了多少分?一个经验丰富的专业软件开发人员平均只得7.8分(满分14分)。这表明,即使像上面这样简单的程序测试,也不是一件容易的事,何况要测试一个具有十多万条语句的空中交通管制系统,一个编译程序,甚至一个普通的工资发放软件呢?可见,软件测试是一项复杂而艰巨的任务,需要系统地学习、训练和实践。

2. 黑盒测试的复杂性分析

黑盒测试是一种常用的软件测试方法,在应用这种方法设计测试用例时,把被测程序看成是一个打不开的黑盒,测试人员在不考虑程序内部结构和内部特性的情况下,只根据需求规格说明,设计测试用例,检查程序的功能是否按照规范说明的规定正确地执行。

如果希望利用黑盒测试方法查出软件中所有的故障,只能采用穷举输入测试。所谓穷举输入测试,就是把所有可能的输入全部用做测试输入。例如,要对Microsoft Windows计算器程序进行测试。检验了1+1等于2后,绝不能保证Windows计算器程序能正确地进行所有的加法运算。很可能当进行1 024+1 024时,计算不正确。由于把被测程序看成了一个黑盒子,所以发现这种问题的惟一途径只能是测试每一种输入情况。

要穷举地测试这个Windows计算器程序,就得考虑所有可能的合法输入。比如,从整数加法开始检测1+1的结果,答案是2,结果正确。然后检测1+3的结果……。因为计算器可以处理32位数字,所以必须测试所有的可能性,直至检测到1+99 999 999 999 999 999 999 999 999 999 999 999为止。这一组数据测试完成后,还需要测试输入2+1,2+2,以此类推。最后输入99 999 999 999 999 999 999 999 999 999 999 999+99 999 999 999 999 999 999 999 999 999 999 999。下一步考虑测试小数:例如,1.0+0.1,1.0+1.1等。为了确保能检查出所有的故障,人们不仅要测试所有合法的输入,而且还要对所有可能的非法输入进行测试。即常规数字相加正确无误之后,还应考虑错误输入是否都得到了相应的处理。例如,按了任意键,输入1+a,Z+I,1a1+2b2等,这样的组合可谓成千上万。同时,经过修改的输入也必须再次进行测试。计算器程序允许输入退格键和删除键,就应该考虑‘5<退格键>7+2=”的情况。前面测试过的每一个数字还要逐个按退格键重新进行测试。此外,还得考虑3个数相加、4个数相加等的情况。输入组合实在太多了,无法进行完全测试,即使使用大型计算机来填数也无济于事。这还仅仅是加法,还有减法、乘法、除法、求平方根、百分数和倒数等。因此,要穷举地测试这个Windows计算器程序,实际上就得给出无穷多个测试用例。

即使这样简单的 Windows 计算器都已使人感到头痛,何况大程序的穷举测试呢。可以设想一下,如果要对 C 编译程序进行黑盒穷举测试,会是怎样的情景。不仅要编制所有合法 C 程序(实际上是个无穷的量)的测试用例,而且还要编制出所有不合法的 C 程序的测试用例,以确保编译程序能检查出这些程序是非法的,例如成功地编译了一个语法有错的程序。对于那些具有“记忆”功能的程序(例如操作系统、数据库系统、航空服务系统等)问题就更严重了。在这类程序中,作业的执行要受以前作业的影响,例如对一个数据的询问、预订某航班飞机票等。这样,人们不仅要检测所有单个合法的、非法的作业,而且还要检测所有可能的作业序列。可见,穷举输入测试是不现实的。

3. 白盒测试的复杂性分析

黑盒穷举测试不现实,那么,另一种常用的测试方法——白盒测试是否可以做到穷举测试呢?白盒测试又称结构测试或基于程序的测试。该方法将被测程序看做一个打开的盒子,允许人们检查其内部结构,测试人员根据程序内部的结构特性,设计、选择测试用例,检测程序的每条路径是否都能按照预定的要求正确地执行。

对于没有学过软件测试的人来说,或许认为使程序中每条路径至少执行一次就做到了穷举测试。然而,程序的路径数目可能是个天文数字。来看一个非常简单的程序,并假定程序中所有判断都是相互独立的,它的控制流图如图 2-5 所示,图中每个结点代表一个语句,每条边或弧则表示两个语句间的控制转移。该图描述了一个由 10~20 条语句构成的程序,其中含有一个重复 20 次的循环语句,而在循环体内,则有一些嵌套的条件语句。那么从 A 点到 B 点的所有路径数有 $5^{20}+5^{19}+\cdots+5^{1}$,其中 5 是贯穿循环体的路径数。大多数人难以想象这么大的数目,可以这样设想 如果每 5 分钟可以写出、执行并验证一个测试用例,那么测试完所有路径大概要花 10 亿年。

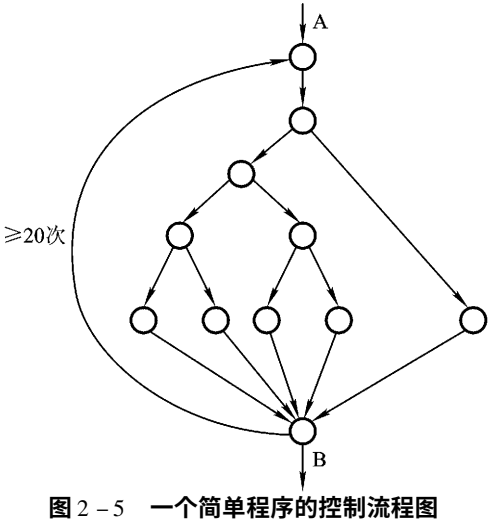


图 2-5 一个简单程序的控制流程图

当然,实际程序中的判断并非都是相互独立的,这意味着可能的实际路径数要比上面估计的小一些。但另一方面,实际程序要比图 2-5 所描述的简单程序规模大得多。因此,举穷路径测试看起来也同穷举输入测试一样是不现实的。

再者,即使程序中每条路径都测试过了,仍不能保证程序没有故障。因为 :第一,穷举路径测

试不能保证程序实现完全符合规格说明的要求。例如,如果要求编写升序排序程序,结果程序被错误地编写成按降序排序。这时,穷举路径测试是毫无用处的;第二,穷举路径测试不可能查出程序中因遗漏路径而出现的错误;第三,穷举路径测试可能发现不了与数据有关的故障。例如,假定程序要求实现:

if (abs(x - y)) < ε

而实际程序则写成:

if (x - y) < ε

…

显然,这个语句有错,但用举穷路径测试,并不一定能发现这个错误。

因此,不论穷举输入测试还是穷举路径测试,都不可能对被测程序进行彻底的测试。E. W. Dijkstra 的一句名言对测试的不彻底性作了很好的注解:"软件测试只能证明故障的存在,但不能证明故障不存在"。

由于穷举测试工作量太大,实际上是行不通的,这就注定了一切实际测试都是不彻底的。因此,软件测试的一个基本问题是经济学问题。软件测试的总目标是充分利用有限的人力和物力资源,高效率、高质量地完成测试。为了降低测试成本,选择测试用例时应注意遵守测试的"经济性"原则:第一,根据程序的重要性和一旦发生故障将造成的损失来确定它的测试等级;第二,认真研究测试策略,以便能开发出尽可能少的测试用例,发现尽可能多的软件故障。

2.4 测试的充分性问题

测试充分性问题是软件测试的另一个重要问题。一位有经验的软件开发管理人员在谈到软件测试时曾说:"不充分的测试是愚蠢的,而过度的测试则是一种罪孽"。其原因在于,不充分的测试势必使软件带着一些未揭露的故障投入运行,这可能使用户承担较大的危险;而过度测试则会浪费许多宝贵的资源。因此,测试的一个合理目标就是:开发出足够的测试用例,以保证软件在典型应用和关键系统中不会存在什么问题。

1. 测试充分性准则

那么什么程度的测试才算充分的呢?这个问题从一般意义上来说可能无法回答,甚至就是针对软件的一个特定部分也不容易回答。要回答这些问题,有许多因素需要考虑,比如,是否关键软件。按照 IEEE/ANSI(美国国家标准学会)的定义,关键软件是指那些失效可能影响到安全或者可能造成巨大经济或社会损失的软件。对于关键软件或者使用范围大、应用方式繁多的软件,很明显,需要许多额外的测试。另外一种观点是考虑软件所涉及领域的常用标准,测试就是验证软件是否与这些标准相一致。比如,在医药生产和家具生产的质量标准方面就存在着明显的不同。

测试充分性准则用来评价一个测试数据集(测试输入数据的集合)按照软件规范说明测试被测软件是否充分。

测试的充分性也可以根据"覆盖率"这一概念进行衡量。覆盖率可以通过两种方式来测定。一种方式是基于软件规格说明,测试检测了其中的多少需求;另一种方式是基于程序源代码,测试检测了其中的多少行代码,多少条语句,或多少条路径等。这两种方法也反映了两种基本的测

试方法——基于规范的测试(黑盒测试)方法和基于结构的测试(白盒测试)方法。

测试充分性准则是在测试之前,由相关各方根据质量、成本和进度等因素规定的,表现为对测试的要求,与软件需求和软件实现有关,具有以下一些基本性质:

- 空测试对于任何软件都是不充分的。
- 对任何软件都存在有限的充分测试数据集,这一性质称为有限性。
- 如果一个测试数据集对一个软件系统的测试是充分的,那么再增加一些测试用例也是充分的,这一性质称为单调性。
- 软件越复杂,需要的测试用例就越多,这一性质称为复杂性。
- 测试得越多,进一步测试所能得到的充分性增长就越少,这一性质称为回报递减律。

2. 测试数据充分性公理

Weyuker 将公理系统应用到软件测试的研究中,给出了几条基于程序的测试数据集充分性公理。

公理 2.1(非外延性公理) 如果有两个功能相同而实现不同的程序,对其中一个是充分的测试数据集对另一个不一定是充分的。

公理 2.2(多重修改公理) 如果两个程序具有相同的语法结构,对一个是充分的测试数据集对另一个不一定是充分的。

两个程序具有相同的语法结构是指,将一个程序中的若干关系运算符、常数和算术运算符用其他关系运算符、常数和算术运算符替换后,得到的另一个语法正确的程序。

公理 2.3(不可分解公理) 对一个程序进行了充分的测试,并不表示对其中的成分都进行了充分的测试。

公理 2.4(非复合性公理) 对程序各单元是充分的测试数据集并不一定对整个程序(集成后)是充分的。

2.5 测试原则

从不同的角度出发,软件测试会派生出两种不同的测试原则。用户希望通过软件测试能充分暴露软件中存在的问题和故障;开发者希望测试能表明软件产品已经正确地实现了用户的需求,没有软件故障存在。因此,软件测试中一个最为重要的问题是人们的心理问题。本节列举出一些至关重要的测试原则或方针,可以视为软件测试和软件开发的“交通规则”或者“生活法则”,有助于透彻了解整个软件测试过程。

1. 完全测试程序是不可能的

理想情况下,测试所有可能的输入,将提供程序行为最完全的信息,但这往往是不可能的。例如,一个程序若有输入量 X 和 Y 及输出量 Z,在字长为 32 的计算机上运行。如果 X、Y 为整数,按功能测试法穷举,测试数据有 $2^{32} \times 2^{32} = 2^{64}$ 个。如果测试一组数据需要 1 ms,一年工作 365×24小时,完成所有测试需 5 亿年。

如果因为某些原因将一些测试输入去掉,比如认为测试条件不重要或者为了节省时间,那么测试就不是完全测试。在实际测试中,完全测试是不可行的,即使最简单的程序也不行,主要有以下几方面的原因:

- 程序输入量太大。
- 程序输出量太多。
- 软件实现途径太多。
- 软件规格说明没有一个客观的标准。从不同的角度看,软件故障的标准可能不同。

这就注定了一切实际测试都是不彻底的。

2. 软件测试是有风险的

如果决定不去测试所有的情况,那就选择了风险。在前面计算器的例子中,如果没有对1 024 + 1 024 = ?进行测试,而碰巧程序对这种情况的处理不正确,那么就留下了一个软件故障在计数器程序中。如果正好用户要计算1 024 + 1 024,这个软件故障就会被发现。这将是一个修复代价很高的软件故障,因为直到软件交付使用时才被用户发现。

不能做到完全测试,不测试又会漏掉一些软件故障。那么测试的目标应该是使有限的测试投资获得最大的收益,即以有限的测试用例检查出尽可能多的软件故障。图2-6说明了测试量和发现的软件故障数量之间的关系。如果试图测试所有的情况,那么费用将大幅度地增加,而漏掉软件故障的数量并不会因费用上涨而显著下降。如果减少测试或者错误地确定测试对象,那么费用很低,但是会漏掉大量软件故障。因此,应学会的一个主要原则是如何把无边无际的可能输入减少到可以控制的范围以及如何针对风险制定出一些明智抉择,去粗存精,找到最合适的测试量,使测试做得不多不少。

3. 测试无法显示隐藏的软件故障

如果要检查一匹马是否感染了寄生虫(bug,软件故障),通过仔细检查,发现了寄生虫存在的迹象,就可以放心地说这匹马感染了寄生虫。

如果对另一匹马进行检查,没有找到寄生虫迹象或者找不到被感染的征兆。也许发现了一些死虫或者废弃的洞穴,但是无法证实有活的寄生虫存在,当然不能说这匹马没有寄生虫。检查的结果只能说明没有发现活的寄生虫存在。软件测试工作与防疫检查工作极为相似,通过测试可以查找并报告发现的软件故障,但是不能保证软件故障全部被找到,也无法报告隐藏的软件故障。继续测试,可能还会发现一些。

4. 存在的故障数量与发现的故障数成正比

现实生活中的寄生虫现象和软件故障几乎一样,两者都是成群出现的。发现一个软件故障

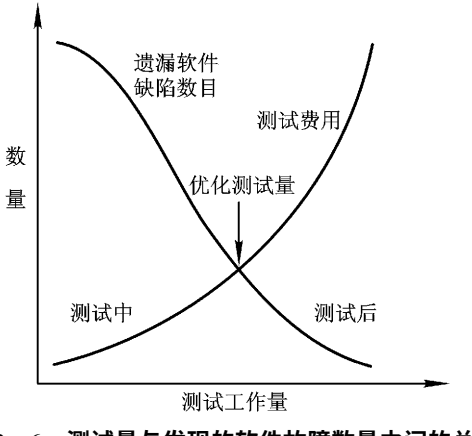


图2-6 测试量与发现的软件故障数量之间的关系

之后,就会接二连三地在附近发现更多的软件故障。在典型程序中,某些程序段看来比其他程序段更容易出错,例如,在 IBM/370 操作系统中,人们注意到一个现象:47% 的软件故障(由用户发现的)只与系统中 4% 的程序模块有关。经验表明,测试后程序中残存的故障数目与该程序中已发现的故障数目成正比,其中原因可能是:

- 程序员怠倦。程序员编写一天代码或许情绪还不错,第二天、第三天可能就会烦躁不安了。一个软件故障很可能是暴露附近更多软件故障的信号。
- 程序员往往犯同样的错误。每个人都有自己的偏好,一个程序员总是反复犯自己容易犯的错误。
- 某些软件故障可能是冰山之巅。某些看似无关的软件故障可能是由一个极其严重的原因造成的。

尽管至今人们还没有对这一现象给出一个令人满意的解释,但这一现象对测试非常有用。根据这一原则,应当对故障集中的程序段进行重点测试。例如,一个含有两个模块 M1 和 M2 的程序,到目前为止在 M1 中发现了 5 个故障,而在 M2 中只发观了一个故障,如果有意不再对 M1 进行更严格的测试,那么由这一原则可知,M1 中含有更多故障的概率要比 M2 更大。这可以让人们更深刻地认识测试过程并采取相应的措施。如果发现某一代码段看起来比其他代码段更容易出错,在试图进一步进行测试时,为了提高测试投资效益,应当花费较多的时间和代价来测试这一代码段。

5. 杀虫剂现象

1990 年 Boris Beizer 在其《软件测试技术》(第 2 版)一书中引用了“杀虫剂现象”一词,用于描述软件测试进行得越多,其程序免疫力越强的现象。这与农药杀虫类似,常用一种农药,害虫最后就有抵抗力,农药发挥不了多大的效力。

为了避免杀虫剂现象的发生,应该根据不同的测试方法开发测试用例,对程序不同部分进行测试,以找出更多的软件故障。

6. 并非所有软件故障都能修复

在软件测试中,令人沮丧的现实是,即使拼尽全力,也不能使所有的软件故障都得以修复。但这并不意味着软件测试没有达到目的,关键是要进行正确的判断、合理的取舍,根据风险分析决定哪些软件故障必须修复,哪些可以不修复。

不修复软件故障的原因可能有:

- 没有足够的时间。软件产品开发中,常常在项目进度中没有为测试留出足够的时间,而软件又必须按时交付。
- 修复风险太大。这种情况很常见。软件本身很脆弱,修复一个软件故障可能导致其他软件故障出现。在紧迫的产品发布和进度压力之下,修改软件将冒很大的风险。在某些情况下,暂时不去理睬软件故障,以避免出现新的软件故障或许是一个可选的安全之道。
- 不值得修复。不常出现的软件故障和在不常用功能中出现的软件故障可以暂不修复。可以躲过和用户有办法预防或避免的软件故障通常也可以不修复。
- 可不算数的软件故障。在某些特殊场合,错误理解、测试错误或者软件规格说明变更可以把软件故障当作附加的功能而不当作故障来对待。

7. 一般不要扔掉测试用例

在使用交互系统进行软件测试时,常常出现这样的情况:一个人坐在计算机前,编写出一些测试用例并用它们对被测程序进行测试。当再次测试程序时(例如,改错后或改进了程序后),就得重新编写测试用例。由于重新编写测试用例需要大量的工作,人们多半要回避它。因此,对程序的重新测试很少能像原来那样严格,这意味着,如果对程序的修改使原先能正确运行的部分出现了故障,那么这个故障常常发现不了。因此,除了真正没有用外,一般不要扔掉测试用例。

8. 应避免测试自己编写的程序

开发和测试是两个不同的活动。开发是创造或者建立一个模块或者整个系统的过程,而测试是为了发现一个模块或者系统中存在故障,不能正常工作的过程。这两个活动之间有着本质的区别。一个人不可能把两个截然对立的角色都扮演好。当一个程序员在完成了设计、编写代码的建设性工作后,要一夜之间改变他的观点,设法对程序形成一个完全否定的态度,那是非常困难的。大部分程序员都不能使自己进入测试状态,揭露自己程序中隐藏的故障,因而大部分程序员不能有效地测试自己的程序。

除了这个心理学问题之外,还有一个重要的问题 程序中可能包含有程序员对问题的叙述或说明的误解而产生的故障。如果是这种情况,当程序员测试自己的程序时,往往还会带着同样的误解进行测试,这样问题很难被发现。可以把测试看做是对一篇论文或一本著作的评审,正如许多作者所知,批评自己的著作是非常困难的。也就是说,找出自己的故障往往是人的心理状态所不容易接受的。

这并不是说程序员不可以测试自己的程序,只是相比之下,如果由他人来进行测试,可能会更有效,更成功。

9. 测试工作应该由独立的专业软件测试机构来完成

独立测试是指软件测试工作由在经济和管理上独立于开发机构的组织进行。独立测试与非独立测势相比具有以下优势:

- 避免软件开发者测试自己开发的软件。由于心理学上的原因,软件开发者难以客观、有效地测试自己的软件,而找出那些因为对问题的误解而产生的软件故障就更加困难。
- 避免软件开发机构测试自己的软件,软件产品开发过程受时间、成本和质量三方面的制约。时间和成本指标便于衡量,而质量却很难度量,因此在软件开发过程中,当时间、成本和质量三者发生矛盾时,质量最容易被忽视,如果测试组织与开发组织来自相同的机构,测试过程就会面临来自与开发组织同一来源的管理方面的压力,使测试过程受到干扰。

采用独立测试方式,无论在技术上还是管理上,对提高软件测试的有效性都具有重要意义。因为独立测试机构具有下面一些优点。

(1) 客观性

对软件测试和软件中可能存在的故障持客观态度,这种客观态度可以解决测试中的心理学问题。经济上的独立性使测试机构有更充分的条件进行测试。

(2) 专业性

软件测试是一个技术含量很高的工作。独立测试作为一种专业测试机构,在长期的工作过程中势必能够积累大量的实践经验,形成自己的专业优势,从而提高测试水平,保证测试质量。

(3) 权威性

由于专业优势,独立测试机构的测试结果更令人信服,评价更客观、公正,更具有权威性。

(4) 资源有保证

独立测试机构的主要任务是进行独立测试工作,这使得测试工作在经费、人力和计划方面更有保证,不会因为开发的压力减少对测试的投入,可以避免开发机构侧重软件开发而忽视测试工作。

10. 软件测试是一项复杂的、具有创造性的和需要高度智慧的挑战性任务

以前,软件产品较小,也不太复杂,即使出现软件故障,也很容易修复,不需付出太多的代价。但是,随着软件规模和复杂性的增加,测试一个大型软件所要求的创造力,可能超过设计那个软件所要求的创造力。现在,生产低质软件的代价太高了,软件行业也发展到强制使用软件测试人员的时代。尽管软件测试不可能发现软件中所有的故障,尽管有一些方法可用来指导测试用例的开发,但使用这些方法仍然需要很大的创造力。

2.6 停止测试的标准

因为无法判定当前发现的故障是否最后一个故障,所以决定什么时候停止测试是一件非常困难的事。受经济条件的限制,测试最终要停止,下面就给出一些实用的停止测试的标准。

2.6.1 五类常用的停止测试标准

在实际工作中,常用的停止测试的标准有 5 类。

第一类标准　测试超过了预定的时间,停止测试。

第二类标准　执行了所有测试用例但没有发现故障,停止测试。

第三类标准　使用特定的测试用例设计方法作为判断测试停止的基础。

第四类标准　正面指出测试停止的要求,比如发现并修改 70 个软件故障。

第五类标准　根据单位时间内查出故障的数量决定是否停止测试。

第一类标准意义不大,因为即便什么都不干也能满足这一条,这不能用来衡量测试的质量。

第二类标准同样也没有什么指导作用,因为它客观上鼓励人们编制查不出故障的测试用例。像上面所讨论的那样,人是有很强工作目的性的。如果告诉测试人员测试用例失败之时就是他完成任务之时,那他会不自觉地以此为目的去编写测试用例,回避那些更有用的、能暴露更多故障的测试用例。

第三类标准把使用特定的测试用例设计方法作为判断测试停止的基础。比如,可以定义测试用例的设计必须满足以下两个条件,作为模块测试停止的标准:

- 条件覆盖准则。
- 边界值分析。

并且由此产生的测试用例最终全部失败。也可以定义满足下面条件时结束测试。这时测试用例产生于:

- 等价类划分。
- 边界值分析。
- 故障猜测。

并且由此产生的测试用例最终全部失败。尽管这类标准比前两个标准优越,但它存在以下3个方面的问题:

- 在没有特定方法的测试阶段中无效,如系统测试阶段。
- 这仍是一个主观的衡量标准,因为无法保证测试人员准确、严格地使用某种测试方法,如边界值分析。
- 这类标准只给出了一个测试用例设计的方法,并不是一个确定的目标。只有测试人员确实能够成功地运用测试用例设计的方法时,才能应用这类标准,并且这类标准只对某些测试阶段适用。

第四类标准正面指出了停止测试的要求,包括两方面问题,具体讨论见2.6.2节。

第五类标准看上去很容易,但在实际使用中要用到很多判断和直觉。它要求人们用图表表示某个测试阶段中单位时间检查出的故障数量,通过分析图表,确定应继续进行测试还是结束这一测试阶段而开始下一测试阶段。例如,假设某一测试阶段发现的故障数如图2-7(a)所示。在第7周,即使这时已找出了预定的故障数,停止测试还是太轻率。因为在第7周中,正处于发现故障的高潮期,明智的决定是继续测试,必要时再设计一些测试用例。另一方面,假设故障数如图2-7(b)所示,在第7周,检查出的故障有明显下降。这时,最好停止测试。当然,还应考虑其他因素的影响,比如,是否由于机器时间不够或合适的测试用例不足所造成了查错效率下降。

2.6.2 第四类停止测试标准

既然测试的目的是找出软件故障,停止测试的标准可以定义为查出某一预定数目的故障。比如,可以定义为某一模块只要找出3个故障就可以停止测试。系统测试的停止标准不妨定义为发现并修改95个故障并至少持续3个月时间。这类标准虽然加强了测试的定义,但仍存在两个问题,一个问题是如何知道将要查出的故障数。为了得到这个数字,要求:

① 估计程序中故障的总数。

② 估计这些故障中通过测试的比例,有多少故障可以很容易地被找出来。

③ 估计哪些故障产生于某些特定的设计过程,估计这些故障将在测试的哪个阶段被检查出。

有几种粗略估计故障总数的方法:

- 根据以往测试程序的经验。

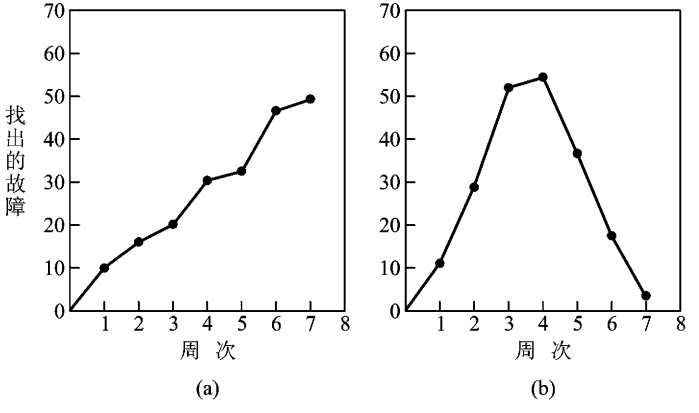


图2-7 单位时间内查出故障数量

• 根据各种故障预测模型。其中有些模型要求首先对软件进行一段时间的测试,记录相继出现的两个故障的时间间隔,然后估计故障总数。另外,一些模型利用故障注入技术,将一些已知的故障注入到被测程序中,然后检查测出的注入故障和非注入故障之比,等等。

• 利用工业界的平均值来获得所要的估计值。比如,一个典型的程序刚编写好时,大概每 100 句中有 4 ~ 8 个故障。

估计通过测试的比例要用到一些主观猜测,也要考虑程序的特性和未知故障可能造成的后果等。

由于不知道故障是怎样发生、何时发生的,所以第三个估计是最难的。美国一家公司的统计表明,在查找出的软件故障中,属于程序编写错误的仅占 36%,属于需求分析和软件设计故障的约占 64%。

举一个简单的例子,如果要测试一个有 10 000 条语句的程序,代码审查之后剩下的故障大约是每 100 句有 5 个,测试的目标是要查出 98% 的代码错,95% 的设计错。估计故障的总数为 500 个,并认为其中有 200 个是代码错,300 个是设计的缺陷。所以这里的目标是查出 206 个代码错,285 个设计错。表 2 – 2 表示了何时查出故障的估计。

表 2 – 2 查出故障的估计

	代码错及逻辑设计错	设计错
单元测试	65%	0
集成测试	30%	60%
系统测试	3%	35%
总计	98%	95%

如果时间进度表是集成测试 3 个月,系统测试 2 个月,那么停止测试的标准可以是:

(1) 只要查出并修改 130 个代码错(200 个代码错的 65%),就可以停止单元测试。

(2) 查出并修改 60 个代码错(200 个代码错的 30%)和 180 个设计错(300 个设计错的 60%),并至少进行 3 个月的集成测试才可以停止测试。如果很快就找出了 240 个故障,说明有可能是低估了故障的总数,所以不能过早地停止集成测试。

(3) 查出并修改 6 个代码错和 105 个设计错,并至少进行 2 个月的系统测试才能停止测试。

第四类标准的另一个问题是过高地估计故障总数。在上例中,如果在测试开始时故障就少于 481 个,若按这一标准则测试永远不会停止。那么是程序写得太好了,没有那么多的软件故障呢,还是测试方法选取不当或测试用例设计不好呢?遇到这种情况,可以请其他测试专家来分析测试用例,判断问题是出在测试用例不足,还是测试用例写得很好,但没有那么多的故障存在。

最好的停止测试标准或许是将上面讨论的几类标准结合起来。因为大部分软件开发项目在单元测试阶段并没有正式地跟踪查错过程,所以这一阶段最好的停止测试标准可能是第一类。对于集成测试和系统测试阶段,停止测试的标准可以是查出了预定数量的故障和达到了一定的测试期限,但还要分析故障 – 时间图,只有当该图指明这一阶段的测试效率很低时才能停止测试。

小结

软件测试是“使用人工或自动手段来运行或测定某个系统的过程,其目的在于检验它是否满足规定的需求或是弄清预期结果与实际结果之间的差别”。该定义包含了两方面的含义:

① 是否满足规定的需求。

② 是否有差别。

这一定义非常明确地提出了软件测试以检验软件是否满足需求为目标。

针对软件测试人员而言,测试的最好定义是:测试以发现故障为目的,是为了发现故障而执行程序的过程。这一定义强调寻找故障是测试的目的。

测试是一种活动,是一个或多个测试用例的集合。测试用例是为特定的目的而开发的一组测试输入、执行条件和预期结果。测试步骤则详细说明了如何设置、执行和评估特定的测试用例。

软件测试主要涉及 5 个方面的问题:

① 谁来执行测试。

② 测试什么。

③ 什么时候测试。

④ 怎样进行测试。

⑤ 停止测试的标准是什么。

软件故障不仅有积累效应,还有放大效应,后期软件故障的修复费用比前期进行相应修改高出 2 ~ 3 个数量级。

不充分的测试是愚蠢的,而过度的测试则是一种罪孽。穷举输入测试和穷举路径测试都是不现实的,这就注定了一切实际测试都是不彻底的。因此,软件测试的总目标是充分利用有限的人力和物力资源,高效率、高质量地完成测试。

软件测试中一个最重要的问题是人们的心理问题,所以,一些至关重要的测试原则或方针必须遵守。

第 2 章习题

1. 什么是软件测试?
2. 软件测试涉及哪几个关键问题?
3. 为什么说软件需求说明是软件故障的最大来源?
4. 简述软件测试的复杂性和经济性。
5. 启动 Windows 计算程序,输入 6,000 – 6 =(逗号不能少),观察计算结果,这是软件故障吗,为什么?
6. 软件测试应遵循哪些重要的原则或方针?
7. 假定无法完全测试某一程序,那么在决定是否应该停止测试时应考虑哪些问题?
8. 为什么完全测试程序是不可能的?
9. 假如星期一测试软件的某一功能时,每小时能发现一个新的软件故障,那么星期二会以什么频率发现软件故障?

第 3 章　软件测试策略

软件的多样性为软件测试带来无限的挑战。每天都有新的软件发布,新的技术不断涌现,产生了许多有趣的测试问题。一个程序员面对各种可能的平台、配置、编程语言和用户,可能会编写出的成百上千行代码,可能需要几十甚至上百个测试人员对其进行测试。所以,软件测试是一个十分艰巨的任务。

为了检验开发的软件是否符合规格说明的要求,测试活动可以采用各种不同的测试手段和策略,如静态测试与动态测试、黑盒测试与白盒测试、自顶向下的集成测试与自底向上的测试以及不同的测试步骤(如单元测试、集成测试、确认测试和系统测试等)。

本章重点:

- 软件生存周期
- 单元测试
- 白盒测试和黑盒测试
- 动态测试和静态测试

3.1　软件开发模型

软件开发模型是软件开发过程、活动和任务的结构框架,它能够清晰、直观地表达软件开发的全部过程,明确规定要完成的主要活动和任务,是软件项目开发的基础。

与任何事物一样,软件也有一个从孕育、诞生、成长到衰亡的生存过程,通常称为软件生存周期,包括制定计划、需求分析、设计、程序编码、测试及运行维护 6 个阶段,即:

第一阶段　计划
第二阶段　需求分析
第三阶段　设计
第四阶段　程序编写
第五阶段　测试
第六阶段　运行和维护

以下给出各阶段的主要任务:

(1) 计划(第一阶段)

确定软件开发的总目标。设想软件的功能、性能、可靠性以及接口等方面的要求。研究完成该项软件任务的可行性,探讨解决问题的方案,对可供开发使用的资源(如计算机软/硬件、人力等)、成本、可取得的效益和开发的进度作出估计,制定完成开发任务的实施计划(Implementation Plan)。

(2) 需求分析(第二阶段)

对开发的软件进行详细的定义。由软件开发人员和用户共同讨论决定哪些需求是可以满足

的并且给出确切的描述,写出软件需求说明或称软件规格说明以及初步的用户手册,提交管理机构审查。

(3) 软件设计(第三阶段)

设计是软件工程的技术核心。在设计阶段应把已确定的各项需求转换成相应的体系结构,在结构中每一组成部分都是功能明确的模块,每个模块体现相应的需求,这一步称为概要设计;在概要设计的基础上进行详细设计,即对每个模块要完成的工作进行具体的描述,包括确定使用的数据结构等,为程序编写打下基础。上述两步设计工作均应写出设计说明,以供后继工作使用并提交审查。

(4) 程序编写(第四阶段)

把软件设计转换成计算机可以接受的程序,即编写出以某种程序设计语言表示的源程序。当然,编写出的程序应该是结构良好、清晰易读并且与设计相一致。

(5) 测试(第五阶段)

测试检验开发的软件是否符合规格说明的要求,它是保证软件质量的重要手段。通常测试工作分为 4 步,即:

① 单元测试 检验各单元模块能否正常工作。

② 集成测试 将已测试的单元模块组装起来进行测试,检验与软件设计相关的程序结构问题。

③ 确认测试 对照软件规格说明,检验开发的软件能否满足所有功能和性能的要求,以决定开发的软件是否合格,能否提交用户使用等。

④ 系统测试 检验开发的软件能否与系统的其他部分(如硬件、数据库、操作人员等)协调工作。

(6) 运行和维护(第六阶段)

已交付的软件投入正式使用以后便进入了运行阶段,这个阶段可能持续若干年,甚至几十年。在运行过程中可能会有多种原因需要对软件进行修改。比如,运行中发现了软件故障,为适应变化了的软件工作环境,为进一步增强软件的功能,提高它的性能等。

以上 6 个阶段表明了软件从开发,直至使用相当长一段时间以后,被新的软件所代替而退役的整个过程。按此顺序逐步转变的过程可用一个软件生存期的瀑布模型加以形象地描述,如图 3 – 1 所示。图中从上到下如同瀑布流水逐级下落,在最后的运行中可能需要多次维护。此外,在实际的项目开发中,为了确保软件的质量,每一步完成以后都要进行复查,及时发现问题并解决问题,以免积压到最后造成修复的更大困难。每一步骤的复查及修改工作用向上的箭头表示。

许多采用瀑布模型的开发组织为有效地实施软件开发,制定了许多软件开发规范或开发标准,明确规定了各个开发阶段应交付的产品及文档,为严格控制软件开发项目的进度,按时交付产品以及保证软件产品的质量创造了有利条件。

瀑布模型多年来广泛流行,它在支持结构化软件开发,控制软件开发的复杂性,促进软件开发工程化等方面起到了显著作用。但是,瀑布模型在大量软件开发实践中也逐渐暴露出了许多缺点,其中最为突出的是该模型缺乏灵活性,无法通过开发活动澄清本来不够确切的软件需求,可能导致开发出的软件并不是用户真正需要的软件,只能进行返工或不得不在维护中纠正需求

的偏差,为此必须付出高额的代价,给软件开发带来了不必要的损失。

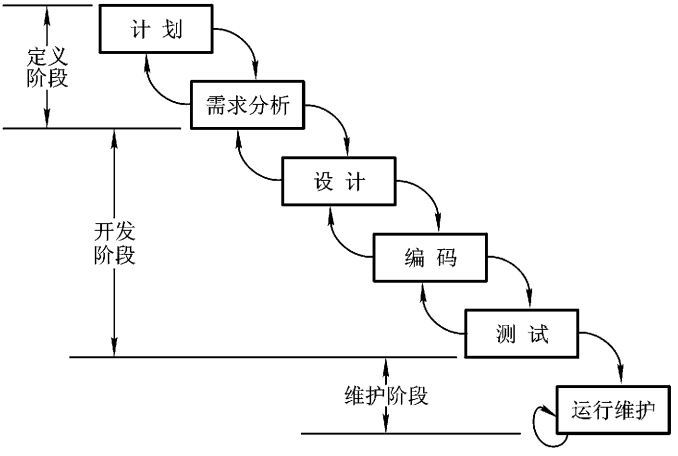


图3-1 软件生存期的瀑布模型

另一种常用的软件开发模型是1988年由TRW公司提出的螺旋模型,如图3-2所示,该模型加入了风险分析。

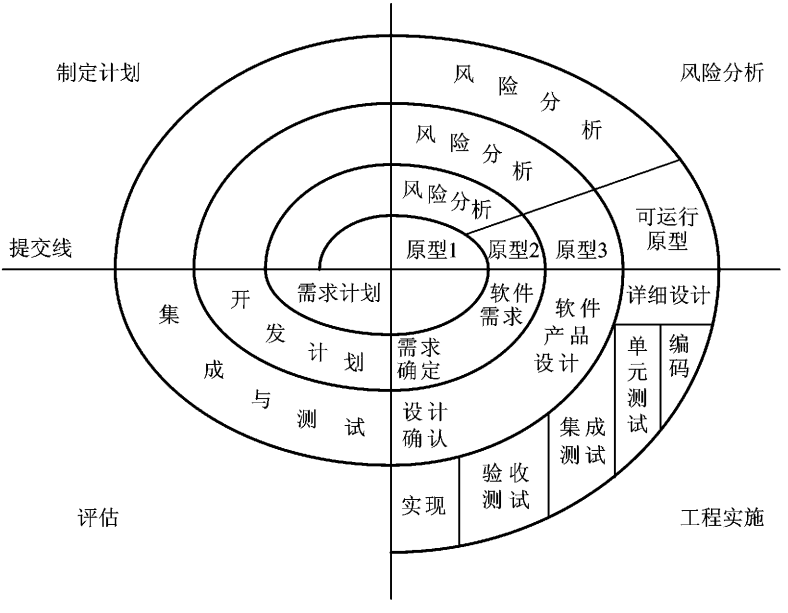


图3-2 软件开发的螺旋模型

在制定软件开发计划时,系统分析员必须回答 项目的需求是什么,需要投入多少人力、物力等资源以及如何安排开发进度等一系列问题。然而若要他们准确无误地回答这些问题是非常困难的,甚至几乎是不可能的,但这又是一个不可回避的问题。凭借经验估计给出初步的设想便带来了一定的风险,因此,风险是任何软件开发项目中普遍存在的问题。不同项目的风险有大有小,实践表明,项目规模越大,问题越复杂,资源、成本、进度等因素的不确定性就越大,承担项目

所冒的风险也就越大。风险是软件开发过程中不可忽视的潜在不利因素,它可能在不同程度上损害到软件开发过程以及软件产品的质量。风险分析的目标就是在造成危害之前,及时对风险进行分析,采取相应的对策,消除或减少风险的损害。

由图 3 – 2 可以看出 :每一螺旋包括 4 个方面的活动,即 :

① 制定计划　确定软件项目开发的目标,选定实施方案,弄清项目开发的限制条件。

② 风险分析　分析所选的实施方案,指出如何识别并降低风险。

③ 实施方案　实施软件开发。

④ 评估方案　评价开发工作,提出修正建议。

每旋转一圈便开发出一个更为完善的软件新版本。例如,在第一圈,确定了初步的目标、方案和限制条件以后,对风险进行识别和分析。如果风险分析表明需求具有不确定性,那么对需求作进一步的修正。

对工程实施做出评价后,给出修正建议,在此基础上再次制定软件开发计划并进行风险分析。在每一圈螺旋线上,风险分析做出是否继续下去的判断。假如风险太大,开发者和用户无法承受,项目有可能被终止。在多数情况下活动会沿螺线继续下去,由内向外逐步延伸,最终得到所期望的系统。

螺旋模型适合于大型软件的开发,该模型的使用需要具有相当丰富的风险评估经验和专门知识。如果项目风险较大,又未能及时发现,势必造成重大损失。

软件测试人员比较喜欢螺旋模式,通过参与最初的设计阶段,项目的来龙去脉比较清楚,可以尽早地了解项目甚至影响项目。测试一直在进行,直到最后宣布成功,不至于在项目末期匆匆忙忙地在短时间内完成测试。

螺旋模型出现较晚,远不如瀑布模型普及,要让开发人员和用户广泛接受,还有待于更多的实践。

3.2 软件测试过程

如同任何产品离不开质量检验一样,软件测试是在软件投入运行前,对软件需求分析、设计规格说明和编码实现的最终审定,在软件生存期中占据着非常突出的重要位置。

很显然,表现在程序中的故障,并不一定是由编码所引起的,很可能是详细设计、概要设计阶段,甚至是需求分析阶段的问题引起的,即使针对源程序进行测试,所发现故障的根源也可能存在于开发前期的各个阶段。解决问题、排除故障也必须追溯到前期的工作。

软件工程界普遍认为 :在软件生存期的每一阶段都应进行评测,检验本阶段的工作是否达到了预期的目标,尽早地发现并消除故障,以免因故障延时扩散而导致后期测试的困难。由此可知,软件测试并不等于程序测试,软件测试应贯穿于软件定义与开发的整个期间。

软件开发是一个自顶向下逐步细化的过程。软件测试则是依相反顺序的自底向上逐步集成的过程。低一级的测试为上一级的测试准备条件。图 3 – 3 表示了软件测试的 4 个步骤,即单元测试、集成测试、确认测试和系统测试。

首先对每一个程序模块进行单元测试,以确保每个模块能正常工作。单元测试大多采用白盒测试方法,尽可能发现并消除模块内部在逻辑和功能上的故障及缺陷,然后,把已测试过的模

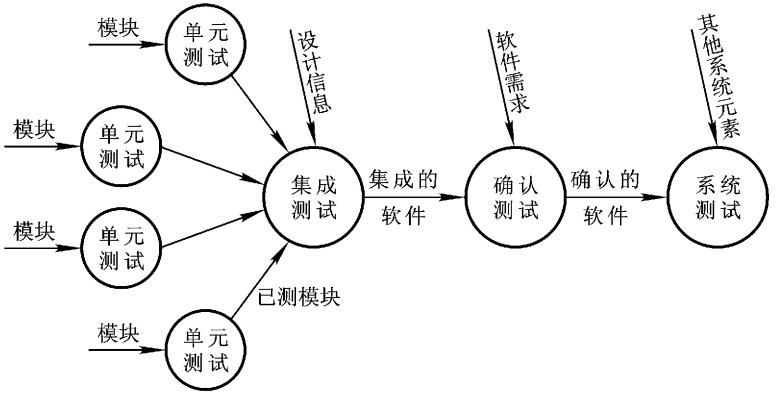


图3-3 软件测试过程

块组装起来,形成一个完整的软件后进行集成测试,以检测和排除与软件设计相关的程序结构问题。集成测试大多采用黑盒测试方法来设计测试用例。确认测试以规格说明规定的需求为尺度,检验开发的软件能否满足所有的功能和性能要求。确认测试完成以后,给出的应该是合格的软件产品,但为了检验开发的软件是否能与系统的其他部分(如硬件、数据库及操作人员)协调工作,还需进行系统测试。

3.2.1 单元测试

单元测试是在软件开发过程中进行的最低级别的测试活动,其测试的对象是软件设计的最小单位。在传统的结构化编程语言中(比如C语言),单元测试的对象一般是函数或子过程。在像C++这样的面向对象的语言中,单元测试的对象可以是类,也可以是类的成员函数。对Ada语言而言,单元测试可以在独立的过程和函数上进行,也可以在Ada包的级别上进行。单元测试的原则同样也可以扩展到第四代语言(4GL)中,这时单元被典型地定义为一个菜单或显示界面。

单元测试又称为模块测试。模块并没有严格的定义,不过按照一般的理解,模块应该具有以下的一些基本属性:

- 名字。
- 明确规定的功能。
- 内部使用的数据或称局部数据。
- 与其他模块或外界的数据联系。
- 实现其特定功能的算法。
- 可被其上层模块调用,也可调用其下属模块进行协同工作。

单元测试的目的是要检测程序模块中有无故障存在,也就是说,一开始并不是把程序作为一个整体来测试,而是首先集中注意力来测试程序中较小的结构块,以便发现并纠正模块内部的故障。单元测试还提供了同时测试多个模块的良机,从而在测试过程中引入了并行性。下面主要来说明单元测试的任务和过程。

1. 单元测试的任务

单元测试针对每个程序模块进行,解决以下 5 方面的问题:

(1) 模块接口测试

模块接口测试是单元测试的基础。只有在数据能够正确地进入、流出的前提下,其他测试才有意义。模块接口测试应该考虑下列一些因素:

- 模块输入参数的个数与形参的个数是否相同。
- 模块输入参数的属性与形参的属性是否匹配。
- 模块输入参数的使用单位与形参的使用单位是否一致。
- 调用其他模块时,实际参数的个数与被调用模块形参的个数是否相同。
- 调用其他模块时,实际参数的属性与被调用模块形参的属性是否匹配。
- 调用其他模块时,实际参数的使用单位与被调用模块形参的使用单位是否一致。
- 调用预定义函数时,所使用参数的个数、属性和次序是否正确。
- 在模块有多个入口的情况下,是否有与当前入口无关的参数引用。
- 是否修改了只作为输入值的形参。
- 各模块对全局变量的定义是否一致。
- 是否把某些常数当做变量来传递等。

如果模块涉及了外部的输入/输出,还应该考虑下列因素:

- 文件属性是否正确。
- OPEN/CLOSE 语句是否正确。
- 格式说明与输入/输出语句是否匹配。
- 缓冲区大小与记录长度是否匹配。
- 文件使用前是否已经打开。
- 文件结束条件是否正确。
- 输入/输出错误处理是否正确。
- 输出信息中是否有文字性错误等。

(2) 模块局部数据结构测试

局部数据结构检查临时存放在模块内的数据在程序执行过程中是否正确、完整,包括内部数据的内容、形式以及相互之间的关系。局部数据结构往往是故障的根源,应注意发现下面的几类错误:

- 不正确或不相容的类型说明。
- 不正确的初始化或缺省值。
- 不正确的变量名,如拼写错或缩写错。
- 下溢、上溢或地址异常等。

除局部数据结构外,单元测试还应检测全局数据对模块的影响。

(3) 模块边界条件测试

检测在数据边界处模块能否正常工作。模块边界测试是单元测试的一个关键任务,很可能发现新的软件故障。实践表明,边界是特别容易出现故障的地方。例如,处理 n 维数组的第 n 个元素时很容易出错,循环执行到最后一次时也可能出错。一些可能与边界有关的数据类型有数值、速度、字符、地址、位置、尺寸、数量等,同时考虑这些边界的第一个/最后一个、最小值/最大

值、最长/最短、最快/最慢、最高/最低、相邻/最远等特征。

(4) 覆盖测试

检测模块运行能否满足特定的逻辑覆盖。

逻辑覆盖要求对被测模块的结构做到一定程度的覆盖。单元测试应对模块中的每一条独立路径进行测试,以检测因计算错误、比较错误和不适当的控制转向而造成的故障。常见的计算错误包括:

- 误用或用错了算符优先级。
- 混合类型运算,例如,实型数和整型数混合运算。
- 初始化错误。
- 计算精度不够。
- 表达式中符号表示错误。

比较判断常与控制流紧密相关,比较错误势必导致控制流错误,因此单元测试还应致力于发现以下错误:

- 不同数据类型的数据进行比较。
- 错误地使用逻辑运算符或优先级。
- 本应相等的数据由于精确度原因而不相等。
- 变量本身有错。
- 循环终止不正确或循环不终止。
- 迭代发散时不能退出。
- 错误地修改了循环控制变量。

(5) 出错处理检测

检测模块出错处理是否有效。程序运行出现异常并不奇怪,良好的设计应该预先估计到各种可能的出错情况,并给出相应的处理措施,使用户遇到这些情况时不至于束手无策。检验程序出错处理也是单元测试的一个任务,对于可能出现的错误处理,应着重检查以下几种情况:

- 运行发生错误的描述是否难以理解。
- 指明的错误与实际遇到的错误是否一致。
- 出错后,是否尚未进行出错处理便引入系统干预。
- 异常处理是否得当。
- 错误描述中是否提供了足够的错误定位信息。

2. 单元测试过程

单元测试一般在编码之后进行。由于每个模块在整个软件中并不是孤立的,在对每个模块进行单元测试时,需要考虑它和周围模块之间的相互联系。为模拟这一联系,在进行单元测试时,必须设置若干个辅助测试模块,这些辅助模块分为两种:

- 驱动模块 用以模拟被测模块的上级模块,相当于被测模块的主程序。
- 桩模块 用以模拟被测模块的下级模块,相当于被测模块调用的子模块。

被测模块与其相关的驱动模块和桩模块共同构成了一个“测试环境”,如图3-4所示。图中设置了一个驱动模块和4个桩模块。驱动模块在单元测试中接受测试数据并将这些数据传递到被测试模块,启动被测模块,并打印出相应的结果。桩模块则由被测模块调用,它们仅作少量

的数据处理,例如打印入口和返回,以便于检验被测模块与其下级模块之间的接口。

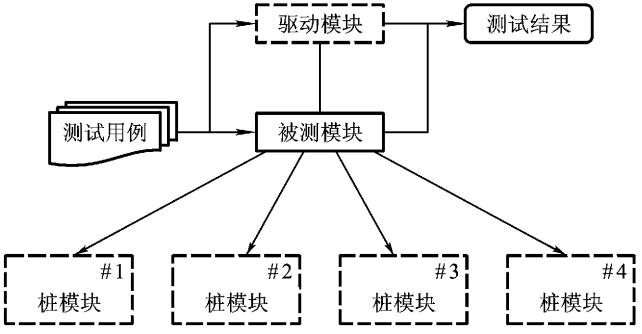


图 3-4 单元测试环境

自然,设计驱动模块和桩模块是一项额外的工作。虽然在单元测试中必须编写这些辅助模块,但它们却不是软件产品的组成部分。如果驱动模块和桩模块比较简单,实际开销相对低些。遗憾的是,有时仅用简单的驱动模块和桩模块并不能完成某些模块的测试任务,特别是桩模块,不能只简单地给出‘曾经进入’的信息。为了能够正确地测试软件,桩模块可能需要模拟实际子模块的功能,这样的桩模块建立就不是很轻松了。

在实际软件开发工作中,单元测试和代码编写所花费的精力大致相同。经验表明:单元测试可以发现很多的软件故障,并且修改它们的成本也很低。在软件开发的后期,发现并修复软件故障将变得更加困难,将花费大量的时间和费用,因此,有效的单元测试是保证全局质量的一个重要部分。在经过测试单元后,系统集成过程将会大大地简化,开发人员可以将精力集中在单元之间的交互作用和全局的功能实现上,而不是陷入充满故障的单元之中不能自拔。

3.2.2 集成测试

时常有这样的情况发生,每个模块都能单独工作,但将这些模块组装起来之后却不能正常工作。程序在某些局部反映不出的问题,很可能在全局上暴露出来,影响到功能的正常发挥,可能的原因有:

- 模块相互调用时引入了新的问题,例如数据可能丢失,一个模块对另一模块可能有不良的影响等。
- 几个子功能组合起来不能实现主功能。
- 误差不断积累达到不可接受的程度。
- 全局数据结构出现错误等。

因此,在每个模块完成单元测试以后,需要按照设计的程序结构图,将它们组合起来,进行集成测试。集成测试是按设计要求把通过单元测试的各个模块组装在一起,检测与接口有关的各种故障。那么,如何组织集成测试呢?是独立地测试程序的每个模块,然后再把它们组合成一个整体进行测试好呢?还是先把下一个待测模块组合到已经测试过的那些模块上去,再进行测试,逐步完成集成好呢?前一种方法称为非增式集成测试法,后一种方法叫做增式集成测试法。

图 3-5 是一个程序例子:图中的 7 个矩形分别表示程序的 7 个模块(子程序或者过程),模

块之间的连线表示程序的控制层次,就是说模块 M1 调用模块 M2、M3 和 M4,模块 M2 调用模块 M5 和 M6 等。非增式测试法的集成过程是:先对 7 个模块中的每一个进行单元测试,可以同时测试或是逐个地测试各个模块,这主要由测试环境(如所用计算机是交互式还是批处理式)和参加测试的人数等情况来决定,然后,在此基础上按程序结构图将各模块连接起来,把连接后的程序当做一个整体进行测试。这种集成测试方法容易出现混乱,因为测试时可能发现一大堆故障,为每个故障定位和纠正非常困难,并且在修复一个故障的同时可能又会引入新的故障,新旧故障混杂,很难断定出错的具体原因和位置。与之相反的另一种集成测试方法是增式集成测试方法。

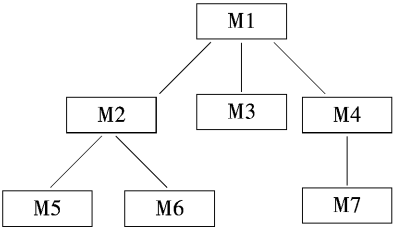


图 3-5 7 个模块的程序简图

增式集成测试是另一种测试方法,它不是孤立地测试每一个模块,而是一开始就把待测模块与已测试过的模块集合连接起来。增量集成测试可以从程序底部开始测试,可以先由 4 个人平行地测试或顺序地测试模块 M3、M5、M6 和 M7,然后测试模块 M2 和 M4,它们不是孤立地测试,而是把模块 M2 连在模块 M5 和 M6 上,模块 M4 连在模块 M7 上。增式集成测试过程,就是不断地把待测模块连接到已测模块集(或其子集)上,对待测模块进行测试,直到最后一个模块(这里是 M1)测试完毕。

关于非增式集成测试和增式集成测试,将在第六章进行详细的介绍。

最后需要说明的是:在软件集成阶段,测试的复杂程度远远超过单元测试的复杂程度。可以类比一下,假设要清洗一台已经完全装配好的食物加工机器,无论喷洒了多少水和清洁剂,一些食物的小碎渣还是会粘在机器的一些死角上,只有任其腐烂并等待以后再想办法。但如果这台机器是拆开的,这些死角也许就不存在或者更容易接触到,并且每一部分都可以毫不费力地进行清洗。

3.2.3 确认测试

集成测试完成以后,分散开发的模块被连接起来,构成了一个完整的程序,其中各模块之间存在的种种问题已被消除,于是进入了确认测试阶段。

所谓确认测试,是对照软件需求规格说明,对软件产品进行评估以确定其是否满足软件需求的过程。比如,编写出的程序相对于软件需求规格说明是否符合要求?程序输出的信息是否用户所要求的信息?程序在整个系统的环境中能否正确稳定地运行?等等,自然包含了对软件需求满足程度的评价。

在软件开发过程中或完成以后,为了对它在功能、性能、接口以及限制条件等方面做出切实的评价,就应进行确认测试。在开发的初期,软件需求规格说明中可能明确规定了确认标准,但在测试阶段需要更详细、更具体地在测试规格说明中加以体现。除了考虑功能、性能以外,还需

检验其他方面的要求。例如,可移植性、兼容性、可维护性、人机接口以及开发的文档资料是否符合要求等。

经过确认测试,应该为已开发的软件做出结论性的评价,这无非存在两种情况:

① 经过检验,软件功能、性能及其他方面的要求都已满足软件需求规格说明的规定,是一个合格的软件。

② 经过检验,发现与软件需求规格说明有相当的偏离,得到了一个缺陷清单,这就需要开发部门和用户进行协商,找出解决的办法。

有关确认测试将在第 7 章中作进一步阐述。

3.2.4 系统测试

软件只是计算机系统的一个重要组成部分,软件开发完成以后,还应与系统中其他部分联合起来,进行一系列系统集成和测试,以保证系统各组成部分能够协调地工作。这里所说的系统组成部分除软件外,还包括计算机硬件及相关的外围设备、数据及采集和传输机构、计算机系统操作人员等。系统测试实际上是针对系统中各个组成部分进行的综合性检验,很接近日常测试实践。例如,在购买二手车时要进行系统测试,在订购在线网络时要进行系统测试等。系统测试的目标不是要找出软件故障,而是要证明系统的性能。比如,确定系统是否满足其性能需求,确定系统的峰值负载条件及在此条件下程序能否在要求的时间间隔内处理要求的负载,确定系统使用资源(存储器、磁盘空间等)是否会超界,确定安装过程中是否会导致不正确的方式,确定系统或程序出现故障之后能否满足恢复性需求,确定系统是否满足可靠性要求等。

系统测试很困难,需要很多的创造性。那么,系统测试应该由谁来进行呢。可以肯定以下人员、机构不能进行系统测试。

- 系统开发人员不能进行系统测试。
- 系统开发组织不能负责系统测试。

之所以如此,第一个原因是,进行系统测试的人必须善于从用户的角度考虑问题,他最好能彻底了解用户的看法和环境,了解软件的使用。显然,最好的人选就是一个或多个用户。然而,一般的用户没有前面所说的各类测试的能力和专业知识,所以理想的系统测试小组应由这样一些人组成:几个职业的系统测试专家、一到两个用户代表、一到两个软件设计者或分析者等。第二个原因是系统测试没有清规戒律的约束,灵活性很强,而开发机构对自己程序的心理状态往往与这类测试活动不相适应。大部分开发软件机构最关心的是让系统测试能按时圆满地完成,并不真正想说明系统与其目标是否一致。一般认为 独立测试机构在测试过程中查错积极性高并且有解决问题的专业知识。因此,系统测试最好由独立的测试机构完成。关于系统测试,有很多种类型,将在第 6 章进一步讨论。

3.2.5 验收测试

验收测试可以类比为建筑的使用者对建筑进行的检测。首先,他认为这个建筑是满足规定的工程质量的,这由建筑的质检人员来保证。使用者关注的重点是住在这个建筑中的感受,包括建筑的外观是否美观,各个房间的大小是否合适,窗户的位置是否合适,是否能够满足家庭的需要等。这里,建筑的使用者执行的就是验收测试。验收测试是将最终产品与最终用户的当前需

求进行比较的过程,是软件开发结束后软件产品向用户交付之前进行的最后一次质量检验活动,它解决开发的软件产品是否符合预期的各项要求,用户是否接受等问题。验收测试不只检验软件某方面的质量,还要进行全面的质量检验并决定软件是否合格。因此,验收测试是一项严格的、正规的测试活动,并且应该在生产环境中而不是开发环境中进行。

验收测试的主要任务包括:

- 明确规定验收测试通过的标准。
- 确定验收测试方法。
- 确定验收测试的组织和可利用的资源。
- 确定测试结果的分析方法。
- 制定验收测试计划并进行评审。
- 设计验收测试的测试用例。
- 审查验收测试的准备工作。
- 执行验收测试。
- 分析测试结果,决定是否通过验收。

如果软件是按合同开发的,合同规定了验收标准,则验证测试由签订合同的用户进行。如果产品不是按合同开发的,开发组织可以采用其他形式的验收测试——Alpha 测试和 Beta 测试。

Alpha 测试和 Beta 测试都是在指定的时间内以生产方式运行并操作软件。Alpha 测试一般在开发公司内,由最终用户进行。被测试的软件由开发人员安排在可控的环境下进行检验并记录发现的故障和使用中的问题。Beta 测试则一般在开发公司之外,由经过挑选的真正用户群进行,它是在开发人员无法控制的环境下,对要交付的软件进行的实际应用性检验。在测试过程中用户要记录遇到的所有问题,并且定期向开发人员通报测试情况。Alpha 测试和 Beta 测试都要求仔细挑选用户,要求用户有使用产品的积极性,能提供良好的硬件和软件配置等。Alpha 测试和 Beta 测试可以分别用做验收测试,不过常常是两者同时都用,一般 Beta 测试在 Alpha 测试之后进行。

验收测试关系到软件产品的命运,因此应对软件产品做出负责任的、符合实际情况的客观评价。制定验收测试计划是做好验收测试的关键一步。验收测试计划应为验收测试的设计、执行、监督、检查和分析提供全面而充分的说明,规定验收测试的责任者、管理方式、评审机构以及所用资源、进度安排、对测试数据的要求、所需的软件工具、人员培训以及其他的特殊要求等。总之,在进行验收测试时,应尽可能去掉一些人为的模拟条件,去掉一些开发者的主观因素,使得验收测试能够得出真实、客观的结论。

3.3 黑盒测试与白盒测试

黑盒测试和白盒测试是两类广泛使用的软件测试方法。

黑盒测试又称功能测试或基于规格说明的测试。

黑盒测试的基本观点是:任何程序都可以看做是从输入定义域映射到输出值域的函数,这种观点将被测程序看做一个打不开的黑盒,黑盒的内容(实现)是完全不知道的,只知道软件要做什么。因无法看到盒子中的内容,所以不知道软件是如何运作的,为什么会这样。但是,很多时

候是可以利用黑盒知识进行有效操作的,例如,大多数人都可以仅凭借黑盒知识成功地操作摩托车。再如前面所述的 Windows 计算器程序,如果输入3.14 159 并按 sqrt 键,就会得到1.772 453 102 341。人们一般不关心计算圆周率的平方根需要经历多少次复杂的运算,只关心它的运算结果是否正确。

在用黑盒测试方法设计测试用例时,测试人员所使用的惟一信息就是软件的规格说明,在完全不考虑程序内部结构和内部特性的情况下,只依靠被测程序输入和输出之间的关系或程序的功能来设计测试用例,推断测试结果的正确性,即所依据的只是程序的外部特性。因此,黑盒测试是从用户观点出发的测试。

白盒测试又称结构测试或基于程序的测试。白盒测试将被测程序看做一个打开的盒子,测试人员可以看到被测的源程序,可以分析被测程序的内部构造,这时测试人员可以完全不考虑程序的功能,只根据其内部构造设计测试用例。

图3-6形象地描述了黑盒测试和白盒测试。

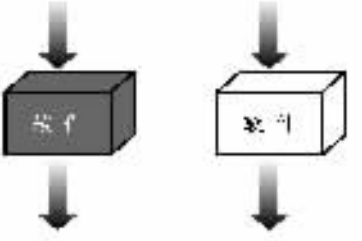


图3-6 黑盒测试和白盒测试

3.3.1 黑盒测试

如上所述,黑盒测试是一类重要的软件测试方法,它根据规格说明设计测试用例,不涉及程序的内部结构。因此,黑盒测试有两个显著的优点:

- 黑盒测试与软件具体实现无关,所以如果软件实现发生了变化,测试用例仍然可以使用。
- 设计黑盒测试用例可以和软件实现同时进行,因此可以压缩项目总的开发时间。

尽管黑盒测试是一类传统的测试方法,有着严格的规定和系统的方式可供参考。但是,在实践中采用黑盒测试也存在一些问题。

一个突出的问题是所谓程序的功能究竟是哪些?众所周知,任何软件作为一个系统都是有层次的。在软件的总体功能之下可能有若干个层次的功能,而测试人员常常只看到低层的功能,他们面临的一个实际问题是在哪个层次上进行测试。如果测试是在高层次上进行,就可能忽略一些细节。如果测试是在低层次上展开,又可能忽视各功能之间存在的相互作用和相互依赖的关系。因此,测试人员需要考虑并且兼顾各个层次的功能。但是,如果为测试人员提供的是一个不分层次的杂乱的规格说明,那么他的黑盒测试工作必定陷入混乱之中,也就不可能取得良好的测试效果。

黑盒测试的另一个问题是功能生成问题。软件开发把原始问题变换成计算机能处理的形式,需要进行一系列的变换,在这一系列变换过程中,每一步都可能得到不同形式的中间成果。

例如,开始时把原始数据变换成表格形式的数据,然后又把表格形式的数据变换成文件上的记录,在此过程中便出现了一系列的功能。首先是填表,然后是输入、输出,再后来又会出现安全保密、口令、恢复及出错处理等功能。

如果软件规格说明是按高层抽象编写的,由于规范本身的高度抽象,不可能涉及许多具体的技术性功能,如文件处理、出错处理等。如果测试用例是根据这样的规格说明得到的,那么实际工程中,详尽的功能测试也可能会遗漏代码中的一些重要部分,因而可能会漏掉其中的一些故障。如果规格说明是按低层抽象编写的,其中必定包含许多技术细节。对于这样的规格说明,用户是非常为难的,因为他们无法理解其中的技术细节,也就无法判断这个规格说明是否反映了他们真正的需求。为了解决这一矛盾,有人建议编写两份规格说明,一份供用户使用,一份供测试人员使用,但即使这样,问题并没有真正解决,因为很难保证这两份规格说明完全一致。

再者,黑盒测试以软件规格说明为依据选取测试数据,其正确性依赖于规格说明的正确性。事实上,人们不能保证规格说明完全正确。很明显,如果程序的外部特性本身有问题或规格说明的规定有误,如规格说明中规定了多余的功能或是漏掉了某些功能,这对于黑盒测试来说是无能为力的。

3.3.2 白盒测试

白盒测试(结构测试)是根据被测程序的内部结构设计测试用例的一类测试,具有很强的理论基础。结构测试要求对被测程序的结构特性做到一定程度的覆盖,或说是“基于覆盖的测试”。测试人员可以严格定义要测试的确切内容,明确提出要达到的测试覆盖率,以减少测试的盲目性,引导测试人员朝着提高测试覆盖率的方向努力,从而找出那些被忽视的程序故障。

语句覆盖是一种最为常见也是最弱的逻辑覆盖准则,它要求设计若干个测试用例,使被测程序的每个语句都至少被执行一次。判定覆盖或分支覆盖则要求设计若干个测试用例,使被测程序的每个判定的真分支和假分支都至少被执行一次。当判定含有多个条件时,可以要求设计若干个测试用例,使被测程序的每个条件的真、假分支都至少被执行一次,这就是条件覆盖。在考虑对程序路径进行全面检验时,可以使用路径覆盖准则。所有这些逻辑覆盖准则将在第5章中进行详细的讨论。

尽管结构测试提供了评价测试的逻辑覆盖准则,但 Howden 认为结构测试是不完全的。理论上,可以构造出一些程序实例证明:每种基于结构的测试最终都将达到极限而不能发现所有的故障。如果程序结构本身有问题,比如程序逻辑有错或者遗漏了某些规格说明已有规定的功能,那么,无论哪一种结构测试,即使其覆盖率达到100%,也是检查不出来的。因此,提高结构的测试覆盖率只能增强对被测软件的信心,但绝不是万无一失的。

3.3.3 黑盒测试与白盒测试的比较

黑盒测试和白盒测试是两种完全不同的测试方法,可以说,它们的出发点不同,并且是两个完全对立的出发点,反映了事物的两个极端。它们各有侧重,都有坚定的拥护者。Robert Poston 认为“白盒测试自20世纪70年代以来一直在浪费测试人员的时间……它不支持良好的软件测试实践,应该从测试人员的工具包中剔除”,而 Edward Miller 则认为“如果能达到85%或更好的分支覆盖率,那么白盒测试能识别出的软件故障,一般是黑盒测试能找出的故障的两倍”。事实

上,黑盒测试和白盒测试在测试实践中都非常有效而且都很实用,不能指望其中的一个能够完全代替另一个。一般而言,在单元测试时大都采用白盒测试,而在确认测试或系统测试中大都采用黑盒测试。如图 3 – 7 所示。

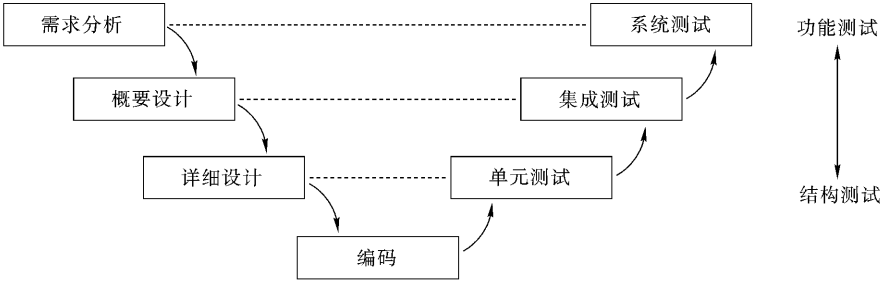


图 3 – 7 功能测试与结构测试

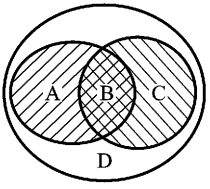
黑盒测试基于外部规格说明,从输入数据与输出数据的对应关系出发设计测试用例,对被测程序的内部情况一无所知,完全不涉及程序的内部结构。很明显,如果外部特性本身有问题或规格说明的规定有误或程序实现了没有被描述的行为(病毒就是这种未描述行为的很好的例子),那么用黑盒测试方法是发现不了的。另一方面,白盒测试完全与之相反,它只根据程序的内部结构进行测试,而不考虑其外部特性。如果程序结构本身有问题,比如说程序逻辑有错误,或是有遗漏,那么用白盒测试则无法发现。如果要求被测软件‘ 做了所有它该做的事,而没有做一点它不该做的事 ”,那么就需要把黑盒测试与白盒测试结合起来使用。因此,两种方法都需要。

表 3 – 1 给出了黑盒测试和白盒测试两类方法的比较。图 3 – 8 则说明了它们各自的能力范围及不足。

表 3 – 1 黑盒测试和白盒测试方法的比较

项 目	白盒测试	黑盒测试
测试依据	根据程序内部结构进行测试	根据软件规格说明设计测试用例
优点	能够对程序内部的特定部位进行覆盖测试	能站在用户立场上进行测试
缺点	① 无法检测程序的外部特性 ② 无法对未实现规格说明的程序部分进行测试	① 不能测试程序内部特定部位 ② 发现不了规格说明的错误
方法	判定覆盖 条件覆盖 判定/条件覆盖 路径覆盖	等价类划分 边界值分析 决策表测试

以上概括地介绍了黑盒测试和白盒测试方法的主要思想,关于黑盒测试和白盒测试的一些主流方法,将在后续章节进行更为详细的讨论。



A	黑盒测试能发现的故障
C	白盒测试能发现的故障
A−B	只能用黑盒测试发现的故障
C−B	只能用白盒测试发现的故障
B	黑盒测试与白盒测试都能发现的故障
D	黑盒测试与白盒测试都不能发现的故障
A+C	用两种测试能发现的故障
A+C+D	软件中的全部故障

图3-8 黑盒测试与白盒测试的比较

3.4 静态测试与动态测试

原则上讲,软件测试方法可以分为两大类 静态测试方法和动态测试方法。

静态测试是指不利用计算机运行被测程序,而是通过其他手段达到检测的目的。动态测试则是指通常意义上的测试——通过运行和使用被测程序,发现软件故障,以达到检测的目的。

模拟这两种测试的最好方法是研究一下汽车的检查过程。踩油门、看车漆、打开前盖检查都属于静态测试技术,而发动汽车、听听发动机的声音、上路行驶则属于动态测试技术。检查软件规格说明属于静态黑盒测试。软件规格说明是书面文档,不是可以执行的程序,因此属于静态测试。软件测试人员可以利用书面文档资料进行静态黑盒测试,认真查找软件缺陷,而检查代码则属于静态白盒测试,它们是在不执行程序的条件下有条理地仔细审查软件设计、体系结构和代码,从而找出软件故障的过程。

静态测试是对被测程序进行特性分析的一些方法的总称。通常在静态测试阶段进行以下一些检测活动:

- 检查算法的逻辑正确性,确定算法是否实现了所要求的功能。
- 检查模块接口的正确性,确定形参的个数、数据类型,顺序是否正确,确定返回值类型及返回值的正确性。
- 检查输入参数是否有合法性检查。如果没有合法性检查,则应确定该参数是否的确不需要合法性检查,否则应加上参数的合法性检查。经验表明,缺少参数合法性检查的代码是造成软件系统不稳定的主要原因之一。
- 检查调用其他模块的接口是否正确,检查实参类型是否正确、实参个数是否正确,返回值是否正确,是否会误解返回值所表示的意思。如果被调用模块出现异常或错误,程序是否添加了适当的出错处理代码。
- 检查是否设置了适当的出错处理,以便在程序出错时,能对出错部分进行重做安排,保证其逻辑的正确性。

- 检查表达式、语句是否正确，是否含有二义性。对于容易产生歧义的表达式或运算符优先级（如，<= 、=、>=、&&、||、++、 --等）可以采用（）运算符以避免二义性。
- 检查常量或全局变量使用是否正确。
- 检查标识符的定义是否规范、一致，变量命名是否能够见名知意，简洁、规范和容易记忆。
- 检查程序风格的一致性、规范性，代码是否符合行业规范，是否所有模块的代码风格一致、规范、工整。
- 检查代码是否可以优化，算法效率是否最高。
- 检查代码是否清晰、简洁和容易理解（注意：冗长的程序并不一定是不清晰的）。
- 检查模块内部注释是否完整，是否正确地反映了代码的功能。错误的注释比没有注释更糟。

静态测试并不是编译程序所能代替的。静态测试可以完成以下工作：

① 可以发现下面的程序缺陷：

- 错用了局部变量和全程变量。
- 不匹配的参数。
- 不适当的循环嵌套和分支嵌套，不适当的处理顺序。
- 无终止的死循环。
- 未定义的变量。
- 不允许的递归。
- 调用不存在的子程序。
- 遗漏了标号或代码。
- 不适当的连接。

② 找到以下问题的根源：

- 未使用过的变量。
- 不会执行到的代码。
- 未引用过的标号。
- 可疑的计算。
- 潜在的死循环。

③ 提供程序缺陷的以下间接信息：

- 所用变量和常量的交叉引用表。
- 标识符的使用方式。
- 过程的调用层次。
- 是否违背编码规则。

④ 为进一步查错作准备。

⑤ 选择测试用例。

⑥ 进行符号测试。

经验表明，使用人工静态测试可以发现大约 30% ~70% 的逻辑设计和编码错误。但是，代码中仍会有大量隐藏的故障无法通过静态测试发现，因此必须通过动态测试进行详细的分析。关于动态测试将在第 4 章和第 5 章进行全面的讨论。

3.5 验证测试与确认测试

软件包括程序以及开发、使用和维护程序所需的所有文档。程序只是软件产品的一个组成部分,表现在程序中的故障,并不一定是由编码所引起的。实际上,软件需求分析、设计和实施阶段都是软件故障的主要来源。因此,软件测试不仅包含对代码的测试,而且包含对软件文档和其他非执行形式的测试。

一种称之为验证的测试就是针对开发过程中的任何中间产品进行的测试。按照IEEE/ANSI的定义,验证测试是为确定某一开发阶段的产品是否满足在该阶段开始时提出的要求而对系统或部分系统进行评估的过程。

所谓验证,是指确定软件开发的每个阶段、每个步骤的产品是否正确无误,是否与其前面开发阶段和开发步骤的产品相一致。验证工作意味着在软件开发过程中开展一系列活动,旨在确保软件能够正确无误地实现软件的需求。有清晰完整的需求吗?有一个好的设计吗?按照设计生产出的产品是什么?验证就是对诸如软件需求规格说明、设计规格说明和代码之类的产品进行评估、审查和检查的过程,属于静态测试。如果是针对代码,其含义就是代码的静态测试——代码评审,而不是动态执行代码。验证测试可应用到开发早期一切可以被评审的事物上,以确保该阶段的产品正是所期望的。

另一种称之为确认的测试则只能通过运行代码来完成。按照IEEE/ANSI的定义,确认测试是在开发过程中或结束时,对系统或部分系统进行评估以确定其是否满足需求规格说明的过程。

所谓确认,是指确定最后的软件产品是否正确无误。比如,编写出的程序与软件需求和用户提出的要求是否符合,或者说程序输出的信息是否用户所要求的信息,这个程序在整个系统的环境中能否正确稳定地运行。正式的确认包括实际软件或仿真模型的运行,确认是‘基于计算机的测试’过程,属于动态测试。

实际上,测试 = 验证 + 确认。将测试分为验证与确认这种分类方法的确认测试包括前述的单元测试、集成测试、确认测试和系统测试。

确认和验证相关联,但也有明显的区别。Boehm是这样来描述两者差别的‘确认要回答的是,我们正在开发一个正确无误的软件产品吗?而验证要回答的是,我们正开发的软件产品是正确无误的吗?”,相应的验证测试计划和确认测试计划涉及不同的内容。

(1)在验证测试计划中要考虑的问题

- 将进行何种验证活动(需求验证、功能设计验证、详细设计验证还是代码验证)。
- 使用的方法(审查、走查等)。
- 产品中要验证的和不要验证的范围。
- 没有验证的部分所承担的风险。
- 产品需优先验证的范围。
- 与验证相关的资源、进度、设备、工具和责任。

(2)在确认测试计划中要考虑的问题

- 测试方法。
- 测试工具。

- 支撑软件(开发和测试共享)。
- 配置管理。
- 风险(预算、资源、进度和培训)。

总之,确认和验证互相补充,保证最终软件产品的正确性、完全性和一致性。

小结

与任何事物一样,软件也有一个从孕育、诞生、成长到衰亡的过程,通常称为软件生存周期,它包括制定计划、需求分析、设计、编写程序、测试及运行维护 6 个阶段。

瀑布模型和螺旋模型是两种广泛流行的软件开发模型。

单元测试是在软件开发过程中进行的最低级别的测试活动,其目的是要检测程序模块中有无故障存在。

集成测试是按设计要求把通过单元测试的各个模块组装在一起,以便检测与接口有关的各种故障。

所谓确认测试,是对照软件需求规格说明,对软件产品进行评估以确定其是否满足需求规格的过程。

系统测试是针对系统中各个组成部分进行的综合性检验,很接近人们的日常测试实践。

验收测试回答开发的软件产品是否符合预期的各项要求,用户是否接受等问题。

黑盒测试和白盒测试是两类广泛使用的软件测试方法。在用黑盒测试设计测试用例时,测试人员所使用的惟一信息就是软件需求规格说明。白盒测试则只根据被测程序的内部结构设计测试用例。

静态测试是指不利用计算机运行被测程序,通过其他手段达到检测程序的目的。动态测试则是指通常意义上的测试——通过运行和使用被测程序,发现软件故障,以达到检测的目的。

验证测试是为确定某一开发阶段的产品是否满足在该阶段开始时提出的要求而对系统或部分系统进行评估的过程。

第 3 章习题

1. 开始编写代码之前有哪些工作要完成?
2. 为什么软件测试人员喜欢螺旋模型?
3. 简述软件测试过程。
4. 单元测试的任务及常用测试方法有哪些?
5. 白盒测试与黑盒测试,静态测试与动态测试,验证测试与确认测试之间有何异同?
6. 如果没有软件规格说明或需求文档,可以进行动态黑盒测试吗,为什么?
7. 如果开发时间紧迫,是否可以跳过单元测试而直接进行集成测试,为什么?

第4章　黑盒测试

黑盒测试是从软件的外部对软件实施测试，也常形容为闭着眼睛测试。本章将介绍几种常用的黑盒测试方法，其中包括等价类划分、边界值分析、决策表测试等。掌握和使用这些方法并不困难，但是，每种方法各有所长，应针对软件开发项目的具体特点，选择合适的测试方法，有效地解决软件开发中的测试问题。

本章重点：

- 等价类划分测试
- 边界值分析
- 决策表测试
- 因果图

4.1　3个被测程序

在第4章和第5章，将采用3个例子来说明各种单元测试方法。一个是三角形问题；一个是逻辑比较复杂的NextDate函数；一个是具有代表性的雇佣金问题。这3个例子合在一起，足可以说明软件测试人员在单元级别上可能会遇到的大多数问题。

4.1.1　三角形问题

三角形问题是软件测试文献中使用最广泛的一个例子。

三角形问题　输入3个整数a、b和c分别作为三角形的3条边，通过程序判断由这3条边构成的三角形类型是　等边三角形、等腰三角形、一般三角形或非三角形（不能构成一个三角形）。有时将直角三角形作为第五类三角形。

可以对三角形问题进行更详细的描述。

三角形问题　输入3个整数a、b和c分别作为三角形的3条边，要求a、b和c必须满足以下条件

Con1　$1 \leqslant a \leqslant 100$

Con2　$1 \leqslant b \leqslant 100$

Con3　$1 \leqslant c \leqslant 100$

Con4　$a < b + c$

Con5　$b < a + c$

Con6　$c < a + b$

程序输出是由这3条边构成的三角形类型　等边三角形、等腰三角形、一般三角形或非三角形。如果输入值不满足这些条件中的任何一个，程序给出相应的信息。例如，“边c的取值不在允许取值的范围内”等。如果a、b和c满足Con1、Con2和Con3，则输出下列4种情况之一：

① 如果不满足条件 Con4、Con5 和 Con6 中的一个,则程序输出为“非三角形”。

② 如果三条边相等,则程序输出为“等边三角形”。

③ 如果恰好有两条边相等,则程序输出为“等腰三角形”。

④ 如果三条边都不相等,则程序输出为“一般三角形”。

显然,这 4 种情况相互排斥。

三角形问题包含了清晰而又复杂的逻辑关系,这正是它经久不衰的主要原因之一。在黑盒测试中,假设开发人员和测试人员知道一些三角形的性质 如任何两边之和大于第三条边等。上限 100 是任意选定的,在利用边界值分析设计测试用例时,将要使用这个上限值。

4.1.2 NextDate 函数

三角形问题之所以复杂,是因为输入与输出之间的关系比较复杂。下面用 NextDate 函数说明另一种复杂性,即输入变量之间逻辑关系的复杂性。

NextDate 函数是一个有 3 个变量 month(月份)、day(日期)和 year(年)的函数。输出为输入日期后一天的日期。例如,如果输入为 1964 年 8 月 16 日,则 NextDate 函数的输出为 1964 年 8 月 17 日。要求输入变量 month、day 和 year 都是整数值,并且满足以下条件:

Con1 $1 \leqslant month \leqslant 12$

Con2 $1 \leqslant day \leqslant 31$

Con3 $1\ 912 \leqslant year \leqslant 2\ 050$

与处理三角形问题一样,也可以使规格说明更具体一些,包括对 day、month 和 year 的无效输入值的响应定义,还可以对无效逻辑组合进行定义。例如对任意年的 2 月 31 日的响应。如果 Con1、Con2 或 Con3 中任何一个条件失效,则 NextDate 都会产生一个输出,指明相应的变量超出了取值范围。例如,“month 值不在 1 ~ 12 范围内”,显然,存在大量的无效 day - month - year 的组合,NextDate 函数将这些组合合并为一个输出“无效输入日期。”

在 NextDate 函数中有两种复杂性来源:一是所讨论输入域的复杂性,一是确定闰年的规则。一年有 365.242 2 天,因此,闰年被用来解决“额外天”的问题。

4.1.3 雇佣金问题

雇佣金问题是一个典型的商务计算例子,包含了计算和决策,因此引出了许多有意思的测试问题。

前亚利桑那州境内的一位步枪销售商销售密苏里州制造的步枪枪机、枪托和枪管。步枪枪机每只卖 45 美元,枪托每只卖 30 美元,枪管每只卖 25 美元。销售商每月至少要售出一支完整的步枪,制造商的生产限额可提供大多数销售商一个月内销售 70 只枪机、80 只枪托和 90 只枪管。每访问一个镇之后,销售商都给密苏里州步枪制造商发封电报,说明那个镇出售的枪机、枪托和枪管数量。当销售商发出售出枪机的个数为“-1”时表明一个月结束,这样步枪制造商就知道当月的销售情况,并计算销售商的雇佣金如下:销售额不到(含)1 000 美元的部分为 10%,1 000(不含)~1 800(含)美元的部分为 15%,超过 1 800 美元的部分为 20%。雇佣金程序生成按月份的销售报告,汇总售出的枪机、枪托和枪管总数,销售商的总销售额以及雇佣金。

这个问题可分为 3 个不同的部分:

① 输入数据部分,用来处理输入数据的有效性。
② 销售额计算部分。
③ 雇佣金计算部分。

4.2 等价类划分测试

穷举测试由于测试输入数量太大,以至于无法实际实现,这就促使测试人员在大量的输入数据中选取一部分作为测试输入。关键在于如何选取测试输入,使得采用这些测试数据能够有效地把隐藏的故障揭露出来。等价类划分是一种典型的黑盒测试方法,该方法完全不考虑程序的内部结构,只根据对软件的要求和说明,即需求规格说明,把程序输入域划分成若干个部分,然后从每个部分中选取少数有代表性的数据作为测试输入。使用等价类划分方法设计测试用例,必须在分析需求规格说明的基础上划分等价类。

4.2.1 等价类划分

等价类划分把程序的输入域划分成若干个互不相交的子集,称之为等价类。所谓等价类是指输入域的某个子集合,所有等价类的并便是整个输入域,这对于测试有两个非常重要的意义:完备性和无冗余性。表示整个输入域提供了一种形式的完备性,而互不相交则可保证一种形式的无冗余性。由于等价类由等价关系决定,因此等价类中的元素有一些共同的特点 如果用等价类中的一个元素作为测试数据进行测试不能发现软件中的故障,那么使用等价类中的其他元素进行测试也不可能发现故障。也就是说,对揭露软件中的故障来说,等价类中的每个元素是等效的。如果测试数据全都从同一个等价类中选取,除去其中一个测试数据对发现软件故障有意义外,使用其余的测试数据进行测试都是徒劳的,它们对测试工作的进展没有任何益处,不如把测试时间花在其他等价类元素的测试中。例如,三角形问题中,如果选择三元组(5,5,5)作为测试输入,可以判定这是一个等边三角形。若再以三元组 (6,6,6)或(100,100,100)作为测试输入,会得到多少新东西呢?直觉告诉我们,这些测试用例会以与测试用例(5,5,5)一样的方式进行,它们具有等价的测试效果,即如果(5,5,5)作为测试数据,能暴露某个软件故障,那么以(6,6,6)或(100,100,100)作为测试数据也能发现这个故障。因此,这些测试用例是冗余的。使用等价类划分测试的目的是既希望进行完备的测试,同时又希望避免冗余。

传统的等价类划分测试的实现分两步进行,一是确定等价类,二是确定测试用例。

1. 划分等价类

软件不能只接收有效的、合理的数据,还应经受意外的考验,即接受无效的或不合理的数据,这样获得的软件才能具有较高的可靠性。因此,在考虑等价类时,应注意区别两种不同的情况。

(1) 有效等价类

有效等价类是指对软件规格说明而言,是有意义的,合理的输入数据所构成的集合。

利用有效等价类,可以检验程序是否实现了规格说明预先规定的功能和性能。在具体问题中,有效等价类可以是一个,也可以是多个。

(2) 无效等价类

无效等价类是指对软件规格说明而言,是不合理或无意义的输入数据所构成的集合。

利用无效等价类,可以检查软件功能和性能的实现是否有不符合规格说明要求的地方。对于具体的问题,无效等价类至少应有一个,也可能有多个。

如何确定等价类,是使用等价类划分方法的一个重要问题。以下给出几条确定等价类的原则:

(1) 按区间划分

如果规格说明规定了输入条件的取值范围或值的数量,则可以确定一个有效等价类和两个无效等价类。例如,如果软件规格说明要求输入的是 1 ~ 12 月中的一个月,则 1 ~ 12 定义了一个有效等价类和两个无效等价类(月 < 1 和月 > 12)。又例如,软件规格说明‘学生允许选修 5 到 8 门课…… ”,则一个有效等价类可取‘选课 5 到 8 门 ”,无效等价类可取‘选课不足 5 门 ’和‘选课超过 8 门 ”。

(2) 按数值划分

如果规格说明规定了输入数据的一组值,而且软件要对每个输入值分别进行处理。则可为每一个输入值确立一个有效等价类,此外针对这组值确立一个无效等价类,它是所有不允许的输入值的集合。

(3) 按数值集合划分

如果规格说明规定了输入值的集合,则可确定一个有效等价类和一个无效等价类(该集合有效值之外)。例如,某软件涉及标识符,要求‘标识符应以字母开头 ”,则‘以字母开头者 ”作为一个有效等价类,“以非字母开头 ”为一个无效等价类。再如,如果输入要求为 TOM、DICK 或 HARRY 这些名字之一,那么可以视为定义了一个有效等价类(采用有效名字之一)和一个无效等价类(比如采用 JOE 这个名字)。

(4) 按限制条件或规则划分

如果规格说明规定了输入数据必须遵守的规则或限制条件,则可以确立一个有效等价类(符合规则)和若干个无效等价类(从不同角度违反规则)。

(5) 细分等价类

等价类中的各个元素在程序中的处理各不相同,则可将此等价类进一步划分成更小的等价类。

在确立了等价类之后,可按表 4 - 1 的形式列出所有划分出的等价类。

表 4 - 1 等 价 类 表

输入条件	有效等价类	无效等价类
…	…	…

同样,也可按照输出条件,将输出域划分成若干个等价类。

2. 设计测试用例

在设计测试用例时,应同时考虑有效等价类和无效等价类测试用例的设计。希望用最少的测试用例,覆盖所有的有效等价类。但对每一个无效等价类,设计一个测试用例来覆盖它即可。

具体来说,根据已列出的等价类表,按以下步骤确定测试用例:

① 为每个等价类规定一个惟一的编号。

② 设计一个新的测试用例,尽可能多地覆盖尚未被覆盖的有效等价类,重复这一步,直到测试用例覆盖了所有的有效等价类。

③ 设计一个新的测试用例,使其覆盖并且只覆盖一个还没有被覆盖的无效等价类。重复这一步,直至测试用例覆盖了所有的无效等价类。

这里规定每次只覆盖一个无效等价类,是因为若用一个测试用例检测多个无效等价类,那么某些无效等价类可能永远不会被检测到,因为第一个无效等价类的测试可能会屏蔽或终止其他无效等价类的测试执行。例如,软件规格说明规定"每类科技参考书50~100册……",若一个测试用例为"文艺书籍10册",在测试中,很可能检测出书的类型错误,而忽略了书的册数错误。

此外,在设计测试用例时,应意识到:预期结果也是测试用例的一个必要组成部分,对采用无效输入的测试也是如此。

等价类划分通过识别许多相等的条件极大地降低了要测试的输入条件的数量。但是,它不测试输入条件组合。

4.2.2 常见的等价类划分测试形式

针对是否对无效数据进行检测,可以将等价类测试分为标准等价类测试和健壮等价类测试。

大多数有关软件测试的教材都讨论了健壮等价类测试的问题,这种形式的测试关注无效数据值,体现了20世纪60—70年代的主流程序设计风格。输入数据检验在当时是一个很重要的问题,"垃圾入,垃圾出"曾经是程序员的格言。对这种问题的通常反应是在程序中加入大量的输入检验代码。很多文献提到,在经典的"输入—处理—输出"结构化的程序体系结构中,输入部分常常占总程序的80%。在这种情况下,强调输入数据的检验很自然也很必要。随着现代程序设计语言的出现,特别是像那些具有强数据类型语言、图形用户界面(GUI)语言的出现,使得输入数据的检验不再那么重要。

为了便于理解,这里以一个有两个输入变量x1和x2的程序F为例,说明标准等价类测试和健壮等价类测试。

假设,输入变量x1和x2在下列范围内取值:

$a \leqslant x2 \leqslant d$,区间[a,b),(b,c),[c,d]

$e \leqslant x2 \leqslant g$,区间[e,f),[f,g]

其中,方括号和圆括号分别表示闭区间和开区间的端点。因此,变量x1和x2的无效等价类分别为:$x1 < a$,$x1 > d$和$x2 < e$,$x2 > g$。

1. 标准等价类测试

标准等价类测试不考虑无效数据值,测试用例使用每个等价类中的一个值,如图4-1所示。

这3个测试用例使用每个等价类中的一个值。事实上,标准等价类测试用例的数量和最大等价类中元素的数目相等。

2. 健壮等价类测试

健壮等价类测试考虑了无效等价类。

① 对有效输入来说,测试用例从每个有效等价类中取一个值(和标准等价类测试一样)。

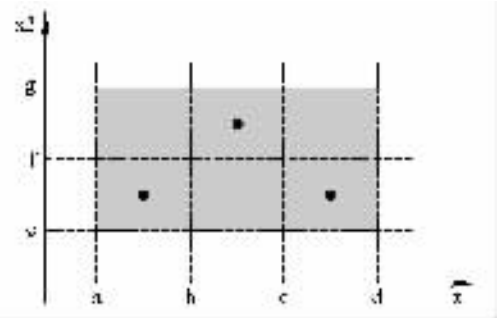


图4-1 标准等价类测试用例

注意,这些测试用例里的每个输入都是有效的。

② 对无效输入来说,一个测试用例有一个无效值,其他值都取有效值。

按照这种策略产生的测试用例如图4-2所示。

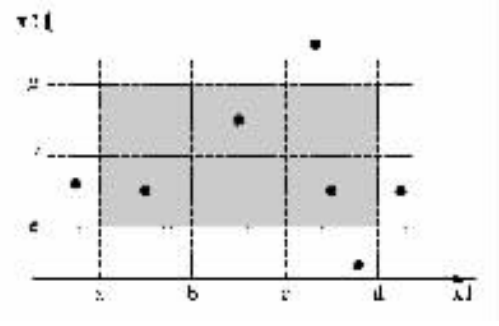


图4-2 健壮性等价类测试用例

健壮等价类测试也有两个问题:一是规格说明往往没有定义无效测试用例的期望输出应该是什么样的,因此,测试人员需要花费大量的时间来定义这些测试用例的期望输出;二是强类型语言没有必要考虑无效输入。传统等价类测试是诸如FORTRAN和COBOL这样的语言占统治地位年代的产物,那时这种无效输入故障很常见。事实上,正是由于经常出现这种错误,才促使人们使用强类型语言。

4.2.3 等价类划分测试举例

1. 三角形问题的等价类测试用例

(1) 传统等价类划分测试用例设计

使用等价类划分方法必须仔细分析程序规范说明。在三角形问题中,输入条件为:

- 整数。
- 3个数。
- 取值在1~100之间。

仔细分析三角形问题,可得出其等价类表,如表4-2所示。根据等价类表,可设计覆盖上述等价类的测试用例。

表 4-2 三角形问题的等价类

	有效等价类	编号	无效等价类	编号
输入3个整数	整数	1	一边为非整数	4
			二边为非整数	5
			三边均为非整数	6
	3个数	2	只有一条边	7
			只有二条边	8
			多余三条边	9
	$1 \leqslant a \leqslant 100$ $1 \leqslant b \leqslant 100$ $1 \leqslant c \leqslant 100$	3	一边为零	10
			二边为零	11
			三边为零	12
			一边 < 零	13
			二边 < 零	14
			三边 < 零	15
			一边 > 100	16
			二边 > 100	17
			三边 > 100	18

测试用例 Test1 = (3,4,5) 便可覆盖有效等价类 1 ~ 3。

覆盖无效等价类的测试用例如表 4-3 所示。

表 4-3 三角形问题的无效等价类测试用例

a	b	c	覆盖的等价类	a	b	c	覆盖的等价类
1.5,	4,	5	4	0,	0,	0	12
3.5,	2.5,	5	5	-3,	4,	6	13
2.5,	4.5,	5.5	6	2,	-7,	-5	14
3			7	-3,	-5,	-7	15
4,	5		8	101,	4,	8	16
2,	3,	4, 5	9	3,	101,	101	17
3,	0,	8	10	101,	101,	101	18
0,	6,	0	11				

(2) 标准和健壮等价类划分测试用例设计

在大多数情况下,从输入域划分等价类,但并不是不能从被测程序的输出域定义等价类,事实上,这对于三角形问题是最简单的方法。

三角形问题有 4 种可能输出 等边三角形、等腰三角形,一般三角形和非三角形。利用这些信息可确定下列输出(值域)等价类。

- R1 = {< a,b,c > 边为 a,b,c 的等边三角形}

- R2 = {<a,b,c> 边为 a,b,c 的等腰三角形}
- R3 = {<a,b,c> 边为 a,b,c 的一般三角形}
- R4 = {<a,b,c> 边 a,b,c 不能形成三角形}

4 个标准等价类测试用例如表 4 -4 所示。

表 4 -4 三角形问题的 4 个标准等价类测试用例

测试用例	a	b	c	预期输出
Test1	5	5	5	等边三角形
Test2	2	2	3	等腰三角形
Test3	3	4	5	一般三角形
Test4	4	1	2	非三角形

考虑 a、b、c 的无效值产生了下面 7 个健壮等价类测试用例,如表 4 -5 所示。

表 4 -5 三角形问题的 7 个健壮等价类测试用例

测试用例	a	b	c	预期输出
Test1	3	4	5	一般三角形
Test2	-1	5	5	a 值不在允许的范围内
Test3	5	-1	5	b 值不在允许的范围内
Test4	5	5	-1	c 值不在允许的范围内
Test5	101	5	5	a 值不在允许的范围内
Test6	5	101	5	b 值不在允许的范围内
Test7	5	5	101	c 值不在允许的范围内

2. NextDate 函数的等价类测试用例设计

NextDate 函数可以很好地说明选择等价关系的技巧。前面已经介绍过,NextDate 是一个含有 3 个变量的函数,即 month、day、year,这些变量的有效等价类可定义如下:

M1 = {month: $1 \leqslant month \leqslant 12$}

D1 = {day: $1 \leqslant day \leqslant 31$}

Y1 = {year: $1912 \leqslant year \leqslant 2050$}

相应的无效等价类是:

M2 = {month: $month < 1$}

M3 = {month: $month > 12$}

D2 = {day: $day < 1$}

D3 = {day: $day > 31$}

Y2 = {year: $year < 1912$}

Y3 = {year: $year > 2050$}

由于有效等价类的数量和程序变量数量相等,因此标准等价类测试用例可以为:

Test1：month = 8，day = 16，year = 1964　　预期输出：1964 年 8 月 17 日

健壮等价类测试中，一个有效测试用例使用每个有效等价类中的一个值，无效测试用例中有一个是无效值，其他都取有效值。健壮测试用例如表 4 – 6 所示。

表 4 – 6　NextDate 函数的健壮等价类测试用例

测试用例	Month	Day	Year	预期输出
Test1	8	16	1964	1964 年 8 月 17 日
Test2	– 1	16	1964	month 不在 1. . 12 中
Test3	13	16	1964	month 不在 1. . 12 中
Test4	8	– 1	1964	day 不在 1. . 31 中
Test5	8	32	1964	day 不在 1. . 31 中
Test6	8	16	1911	year 不在 1912. . 2050
Test7	8	16	2051	year 不在 1912. . 2050

如果更仔细地选择等价关系，得到的等价类可能会更有用。

例如，如果 day 不是某月的最后一天，NextDate 函数只需对 day 值直接加 1 就可以了。对于月末，day 需重新被置为 1，month 加 1。到了年末，day 和 month 都要重新置为 1，year 加 1。如果这一年是闰年，如何确定有关 month 的最后一天呢？经过分析之后，可以生成下面的等价类：

M1 = {month： month 有 30 天}

M2 = {month： month 有 31 天}

M3 = {month： month 是二月}

D1 = {day： 1 ≤ day ≤ 28}

D2 = {day： day = 29}

D3 = {day： day = 30}

D4 = {day： day = 31}

Y1 = {year： year = 2000}

Y2 = {year： year 是闰年}

Y3 = {year： year 不是闰年}

通过选择有 30 天的月份和有 31 天的月份这两个单独的类，可以简化月份最后一天这个问题。通过把 2 月单独作为一个类，可以更多地注意闰年问题。这里要特别注意 day 值，将 day 值分成 4 个独立的等价类 D1、D2、D3 和 D4，其中 D1 中的 day（几乎）总是加 1，而 D4 中的 day 只对 M2 中的 month 有意义。最后，将 year 值分成 3 个等价类，包括 2000 年这个特例、闰年和非闰年类。这并不是一个完美的等价类划分，但是通过这种等价类划分可以发现更多隐藏的故障。

机械地从对应等价类中选择输入值而不考虑其应用领域的相关知识，譬如没有考虑两个不能同时出现的取值。自动测试数据生成经常会遇到这类问题，因为领域知识不是通过等价类选择能够获得的。健壮等价类测试用例如表 4 – 7 所示。标准等价类测试用例与健壮等价类测试用例的前四个一样，即 Test1、Test2、Test3 和 Test4。

表 4－7 NextDate 函数另一种划分的健壮等价类测试用例

测试用例	Month	Day	Year	预期输出
Test1	4	15	2000	2000 年 4 月 16 日
Test2	5	29	1996	1996 年 5 月 30 日
Test3	2	30	2002	2002 年 2 月 31 日(不可能的日期)
Test4	4	31	2000	2000 年 5 月 1 日(不可能的输入)
Test5	－1	15	1996	month 不在有效值域 1..12 中
Test6	13	29	2002	month 不在有效值域 1..12 中
Test7	4	－1	2000	day 不在有效值域 1..31 中
Test8	5	32	1996	day 不在有效值域 1..31 中
Test9	2	30	1911	year 不在有效值域 1912..2050 中
Test10	6	31	2051	year 不在有效值域 1912..2050 中

3. 雇佣金问题的等价类测试用例设计

雇佣金问题的输入定义域,由于受枪机、枪托和枪管的限制,很自然地被划分为 3 类,每一类有一个有效等价类和两个无效等价类。

有效等价类如下:

L1 = {枪机: $1 \leqslant$ 枪机 $\leqslant 70$}

L2 = {枪机 = -1}

S1 = {枪托: $1 \leqslant$ 枪托 $\leqslant 80$}

B1 = {枪管: $1 \leqslant$ 枪管 $\leqslant 90$}

无效等价类如下:

L3 = {枪机 枪机 = 0 或枪机 < -1}

L4 = {枪机 枪机 > 70}

S2 = {枪托 枪托 < 1}

S3 = {枪托 枪托 > 80}

B2 = {枪管 枪管 < 1}

B3 = {枪管 枪管 > 90}

由于变量枪机也被用来表示‘不再有电报’的信息,即当枪机为 -1 时,While 循环终止,因此可定义另一个有效等价类 L2。

根据有效等价类,可以设计出一个标准等价类测试用例:

Test1 枪机 = 4,枪托 = 5,枪管 = 9

7 个健壮等价类测试用例如表 4－8 所示。

表 4-8 雇佣金问题的健壮等价类测试用例

测试用例	枪机	枪托	枪管	预期输出
Test1	4	5	9	55.5
Test2	-1	45	55	枪机值不在允许的范围内
Test3	71	45	55	枪机值不在允许的范围内
Test4	35	-1	55	枪托值不在允许的范围内
Test5	35	81	55	枪托值不在允许的范围内
Test6	35	45	-1	枪管值不在允许的范围内
Test7	35	45	91	枪管值不在允许的范围内

无论是标准等价类还是健壮等价类测试用例,都只有一个是有效的等价类测试用例,很难判断雇佣金问题的计算部分有没有问题,因此,输入域等价类划分不能产生令人满意的测试用例集合。对雇佣金问题的输出值域定义等价类也许可以改进测试用例集合。

销售额是已售枪机、枪托和枪管的函数,即

销售额 = 45 × 枪机 + 30 × 枪托 + 25 × 枪管

可以在佣金值域上定义 3 个等价类:

S1 = {< 枪机,枪托,枪管 > : 销售额 ≤ 1000}

S2 = {< 枪机,枪托,枪管 > : 1000 < 销售额 ≤ 1800}

S3 = {< 枪机,枪托,枪管 > : 销售额 > 1800}

根据输出域等价类设计测试用例如表 4-9 所示。

表 4-9 雇佣金问题的输出域等价类测试用例

测试用例	枪机	枪托	枪管	销售额	雇佣金
Test1	5	5	5	500	50
Test2	15	15	15	1500	175
Test3	25	25	25	2500	360

这些测试用例与健壮等价类测试用例结合起来,可以很好地测试雇佣金问题。

4.2.4 等价类划分测试的指导方针

使用等价类划分测试时,应注意以下几点:

- 如果实现的语言是强类型语言(无效值会引起运行时出错),则没有必要使用健壮等价类测试。
- 如果错误输入检查非常重要,则应进行健壮等价类测试。
- 如果输入数据以离散值区间或集合的形式定义,则等价类测试是合适的,当然也适用于变量值越界会造成故障的系统。
- 在发现合适的等价关系之前,可能需要进行多次尝试,就像 NextDate 函数一样。如果不能肯定存在‘明显’或‘自然’的等价关系,最好对任何合理的实现进行再次预测。

4.3 边界值分析

人们从长期的测试工作经验得知,大量的故障往往发生在输入定义域或输出值域的边界上,而不是在其内部。说明边界值分析、测试的最佳方式是,如果在悬崖峭壁边上可以自如行走,那么在平地上行走就更没有问题了。但是,在软件设计和程序编写中,常常对规格说明中的输入域边界或输出域边界重视不够,以致形成一些差错。实践表明,在设计测试用例时,对边界附近的处理必须给予足够的重视。为检验边界附近的处理设计专门的测试用例,常常可以取得良好的测试效果。

使用边界值分析方法设计测试用例,首先应确定边界情况。输入等价类与输出等价类的边界,就是应着重测试的边界情况。边界值分析方法的基本思想是,选取正好等于,刚刚大于或刚刚小于边界的值作为测试数据,而不是选取等价类中的典型值或任意值作为测试数据。

边界值分析方法是一种最有效的黑盒测试方法,但当边界情况很复杂的时候,要找出适当的边界测试用例还需针对问题的输入域、输出域边界,耐心细致地逐个进行考虑。

4.3.1 边界条件

边界条件是一些特殊情况。程序在处理大量中间数值时都正确,但在边界处可能出现错误。比如,在作三角形判断时,要输入三角形的 3 条边长 a、b 和 c。我们知道:当满足 a + b > c、a + c > b 及 b + c > a 时才能构成三角形。但如果把 3 个不等式中的任何一个大于号“ > ”错写成大于等于号“ ≥ ”,那就无法构成三角形了。问题恰恰出现在那些容易被疏忽的边界上。再如,下面是一个极简单的程序。

```
/* Create a 10 element integer array */
/* Initiatize each element to -1 */
main ()
 {
      int data [10] ;
      int i ;
      For (i = 1 ;10 ;i ++)
      data(i) = -1 ;
}
```

这段代码的意图是创建包含 10 个元素的数组并为数组中的每一个元素赋初值 -1,它建立了包含 10 个整数的数组 data 和一个计数变量 i。For 循环从 1 ~ 10,数组中从第 1 个元素到第 10 个元素被赋予数值 -1,边界问题在哪儿?

在大多数计算机编程语言中,当定义数组时,第一个创建的元素是 0,而不是 1。该程序实际上创建了一个从 data(0) 到 data(10) 共 11 个元素的数组。循环从 1 ~ 10 将数组元素初始化为 -1,但是数组的第一个元素是 data(0),它没有被初始化。程序执行完毕,数组值如下:

data(0) = 0 ; data(1) = -1 ; data(2) = -1 ; data(3) = -1 ; data(4) = -1
data(5) = -1 ; data(6) = -1 ; data(7) = -1 ; data(8) = -1 ; data(9) = -1

data(10) = -1

此时 data(0) 的值是 0,而不是 -1。如果程序员以后忘记了或者其他程序员不知道这个数组只对 data(1) ~ data(10) 进行了初始化,那么他就可能会用到数组的第 1 个元素 data(0),以为它的值是 -1。诸如此类的问题很常见。

刚开始时,可能意识不到一组给定数据包含了多少边界,但是仔细分析总可以找到一些不明显的、有趣的或可能产生软件故障的边界。实际上,边界条件就是软件操作界限所在的边缘条件。

一些可能与边界有关的数据类型有 数值、速度、字符、地址、位置、尺寸、数量等。同时,考虑这些数据类型的下述特征:

第一个/最后一个、最小值/最大值、开始/完成、超过/在内、空/满、最短/最长、最慢/最快、最早/最迟、最高/最低、相邻/最远等。

这是一些可能出现的边界条件。每一个软件测试问题都不相同,可能包含各式各样的边界条件,应视具体情况而定。

4.3.2 次边界条件

上面讨论的边界条件比较容易发现,它们在软件规格说明中或者有定义,或者可以在使用软件的过程中确定。有些边界在软件内部,用户几乎是看不到,但软件测试仍有必要对这些边界条件进行检查,这样的边界条件称为次边界条件或者内部边界条件。2 的幂次方和 ASCII 码就是这种例子。

寻找次边界条件比较困难,虽然不要求软件测试人员成为程序员或者具有阅读源代码的能力,但要求软件测试员能大体了解软件的工作方式。

1. 2 的幂次方

计算机中的数是用二进制数‘0’和‘1’来表示的,8 位二进制数组成一个字节,4 个字节组成一个字。表 4 - 10 列出了常用的 2 的幂次方及对应十进制数的表示范围或值。

表 4 - 10 2 的幂次方所表示的范围

术 语	范 围
位	0 或 1
半字节	0 ~ 15
字节	0 ~ 255
字	0 ~ 65 535 或 0 ~ 4 294 967 295
千位	1 024
兆位	1 048 576
亿位	1 073 741 824

表 4 - 10 所列的范围是作为边界条件的重要数据。但它们通常在软件内部使用,外部是看不见的,除非用户提出这些范围,否则在软件需求规格说明中不会明确指出。

进行软件测试时,要考虑是否需要对这些边界条件进行测试。例如,假设某种通信协议支持

256 条命令,为了提高数据传输效率,通信软件总是将常用的信息压缩到一个很小的单元中,必要时再扩展为大一些的单元。比如将常用的 15 条命令压缩为一个半字节数据,在遇到第 16 ~ 256 之间的命令时,软件转而发送一个一字节的命令。用户只知道可以执行 256 条命令,并不知道软件根据半字节/字节边界执行了不同的计算和操作。为了覆盖所有可能的 2 的幂次方次边界,还要考虑临近半字节边界的 14、15 和 16,以及临近字节边界的 254、255 和 256。

2. ASCII 表

另一个常见的次边界条件是 ASCII 字符表。表 4 - 11 给出了部分 ASCII 字符。

表 4 - 11 部分 ASCII 值

字 符	ASCII 值	字 符	ASCII 值
Null	0	B	66
space	32	Y	89
/	47	Z	90
0	48	[	91
1	49	‘	96
2	50	a	97
9	57	b	98
:	58	y	121
@	64	z	122
A	65	{	123

表 4 - 11 中,数字 0 ~ 9 的 ASCII 值是 48 ~ 57。斜杠字符(/)在数字 0 的前面,而冒号字符(:)在数字 9 的后面。大写字母 A ~ Z 对应的 ASCII 码值是 65 ~ 90。小写字母对应的 ASCII 码值是 97 ~ 122。这些情况都表示次边界条件。

如果对文本输入或文本转换软件进行测试,在考虑数据区间包含哪些值时,最好参考一下 ASCII 表。例如,如果测试的文本框只接受用户输入字符 A ~ Z 和 a ~ z,就应该在非法区间中,检测 ASCII 表中位于这些字符前后的值——@、[、‘ 和 {。

4.3.3 边界值分析测试

为了便于理解,这里讨论一个有两个变量 x1 和 x2 的程序 F,假设输入变量 x1 和 x2 在下列范围内取值:

$$a \leqslant x1 \leqslant b$$

$$c \leqslant x2 \leqslant d$$

区间[a,b]和[c,d]是 x1 和 x2 的值域,强类型语言允许显式地定义这种变量值域。事实上,边界值测试更适于采用非强类型语言编写的程序。程序 F 的输入空间(定义域)如图 4 - 3 所示。带阴影矩形中的任何点都是程序 F 的有效输入。

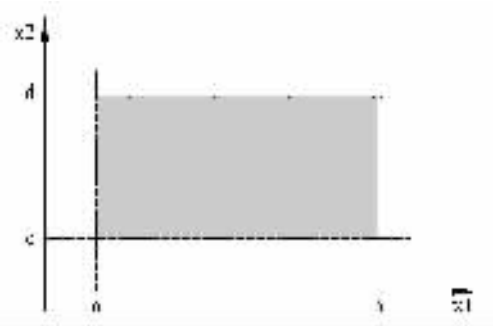


图4-3 两个变量程序的输入域

边界值分析测试的基本原理是故障往往出现在输入变量的边界值附近。例如,当循环条件本应当判断“≤”时,却错写成了“<”,计数器常常少记一次等。美国陆军(CECOM)对其软件进行了研究,发现有相当一部分的故障是由边界值引起的。

边界值分析基于一种在可靠性理论中称为“单故障”的假设,即由两个(或两个以上)故障同时出现而导致软件失效的情况很少,也就是说,软件失效是由单故障引起的。边界值分析利用输入变量的最小值(min)、稍大于最小值(min+)、域内任意值(nom)、稍小于最大值(max-)和最大值(max)来设计测试用例,即通过使所有变量取正常值,只使一个变量分别取最小值、略高于最小值、略低于最大值和最大值。有两个输入变量的程序F的边界值分析测试用例是(如图4-4所示):

$$\{<x1_{nom},x2_{min}>,<x1_{nom},x2_{min}+>,<x1_{nom},x_{2nom}>,<x1_{nom},x2_{max}>,<x1_{nom},x2_{max}->,$$
$$<x1_{min},x2_{nom}>,<x1_{min}+,x2_{nom}>,<x1_{max},x2_{nom}>,<x1_{max}-,x2_{nom}>\}$$

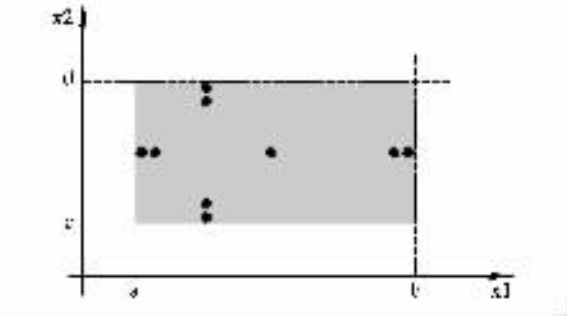


图4-4 有两个变量程序的边界值分析测试用例

对于一个含有n个变量的程序,保留其中一个变量,让其余变量取正常值,这个被保留的变量依次取值min、min+、nom、max-和max,对每个变量重复进行,这样,对于一个n变量的程序,边界值分析测试会产生4n+1个测试用例。

不管采用什么语言,变量的min、min+、nom、max-、max值根据语境可以清楚地确定。例如雇佣金问题中的变量,其min、min+、nom、max-、max值很容易确定。如果没有显式地给出边界,

例如三角形问题,可以人为地设定一个边界。显然,边长的下界是 1(边长为负没有什么意义),但如何来确定上界呢?在默认情况下,可以取最大可表示的整型值(某些语言里称为 MAXINT),或者规定一个数作为上界,如 100 或 1 000。

4.3.4 健壮性测试

健壮性测试是边界值分析测试的一种扩展,变量除了取 min、min+、nom、max-、max 5 个边界值外,还要考虑采用一个略超过最大值(max+)以及一个略小于最小值(min-)的取值,看看超过极限值时系统会出现什么情况。健壮性测试用例如图 4-5 所示。

边界值分析的大部分讨论都可直接用于健壮性测试。健壮性测试最有意义的部分不是输入,而是预期的输出。观察例外情况如何处理,例如,当物理量超过其最大值时会出现什么情况?如果是飞机机翼的迎角超过其最大值,则飞机可能失控 如果是公共电梯的负荷能力超过其最大值,可能出现可怕的情形。对于强类型语言,健壮性测试可能比较困难。例如在 C 语言中,如果变量被定义在特定的范围内,则超过这个范围的取值都会产生导致正常执行中断的故障。

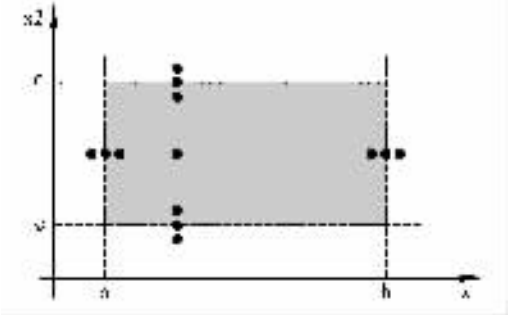


图 4-5 两个变量程序的健壮性测试用例

4.3.5 边界值分析举例

前面 3 个例子的每一个都是一个含有 3 个输入变量的函数。给出每个例子的所有边界值分析测试用例非常浪费空间,所以,下面只列出一部分测试用例。

1. 三角形问题的边界值分析测试用例设计

在三角形问题描述中,除了要求边长是整数外,没有给出其他的限制条件。显然,边长下界为 1,边长上界可取为 100。表 4-12 给出了其边界值分析测试用例。

表 4-12 三角形问题的边界值分析测试用例

测试用例	a	b	c	预期输出
Test1	50	50	1	等腰三角形
Test2	50	50	2	等腰三角形
Test3	50	50	50	等边三角形
Test4	50	50	99	等腰三角形
Test5	50	50	100	非三角形

续表

测试用例	a	b	c	预期输出
Test6	50	1	50	等腰三角形
Test7	50	2	50	等腰三角形
Test8	50	99	50	等腰三角形
Test9	50	100	50	非三角形
Test10	1	50	50	等腰三角形
Test11	2	50	50	等腰三角形
Test12	99	50	50	等腰三角形
Test13	100	50	50	非三角形

2. NextDate 函数的边界值分析测试用例设计

在 NextDate 函数中，规定了变量 month、day、year 相应的取值范围，即 $1 \leqslant month \leqslant 12$，$1 \leqslant day \leqslant 31$ 和 $1912 \leqslant year \leqslant 2050$。表 4 – 13 给出了其健壮性测试用例。

表 4 – 13 NextDate 函数健壮性测试用例

测试用例	Month	Day	Year	预期输出
Test1	6	15	1911	1911 年 6 月 16 日
Test2	6	15	1912	1912 年 6 月 16 日
Test3	6	15	1913	1913 年 6 月 16 日
Test4	6	15	1975	1975 年 6 月 16 日
Test5	6	15	2049	2049 年 6 月 16 日
Test6	6	15	2050	2050 年 6 月 16 日
Test7	6	15	2051	2051 年 6 月 16 日
Test8	6	– 1	1975	day 不在有效值域 1..31 中
Test9	6	1	1975	1975 年 6 月 2 日
Test10	6	2	1975	1975 年 6 月 3 日
Test11	6	30	1975	1975 年 7 月 1 日
Test12	6	31	1975	不可能的输入日期
Test13	6	32	1975	day 不在有效值域 1..31 中
Test14	– 1	15	1975	month 不在有效值域 1..12 中
Test15	1	15	1975	1975 年 1 月 16 日
Test16	2	15	1975	1975 年 2 月 16 日
Test17	11	15	1975	1975 年 11 月 16 日
Test18	12	15	1975	1975 年 12 月 16 日
Test19	13	15	1975	month 不在有效值域 1..12 中

3. 雇佣金问题的边界值分析测试用例设计

对雇佣金问题,考虑输出域的边界值,尤其是在 $1 000 和 $1 800 这两个临界点附近。用输出值域来确定测试用例,部分原因是因为在不同的销售额范围内,雇佣金按不同比例计算,想找出边界值在 $100、$1 000、$1 800 以及 $7 800 上的输入变量组合。最小值 $100(枪机 = 1,枪托 = 1,枪管 = 1)和最大值 $7 800(枪机 = 70,枪托 = 80,枪管 = 90)对应的输入很容易确定,测试用例 9(枪机 = 10,枪托 = 10,枪管 = 10)是$1 000 对应的输入。如果调整输入变量,就能得到稍小于和稍大于边界的值(测试用例 6 ~ 8 和 10 ~ 12)。这实际上可以认为是一种特殊值测试,因为利用了数学洞察力来产生测试用例。

表 4 - 14 列出了其输出边界值分析的测试用例。

表 4 - 14 雇佣金问题输出边界值分析测试用例

测试用例	枪机	枪托	枪管	销售额	雇佣金	预期输出
Test1	1	1	1	100	10	最小输出值
Test2	1	1	2	125	12.5	稍大于最小输出值
Test3	1	2	1	130	13	稍大于最小输出值
Test4	2	1	1	145	14.5	稍大于最小输出值
Test5	5	5	5	500	50	中间值
Test6	10	10	9	975	97.5	稍小于边界值
Test7	10	9	10	970	97	稍小于边界值
Test8	9	10	10	955	95.5	稍小于边界值
Test9	10	10	10	1000	100	边界值
Test10	10	10	11	1025	103.75	稍大于边界值
Test11	10	11	10	1030	104.5	稍大于边界值
Test12	11	10	10	1045	106.75	稍大于边界值
Test13	14	14	14	1400	160	中间值
Test14	18	18	17	1775	216.25	稍小于边界值
Test15	18	17	18	1770	215.5	稍小于边界值
Test16	17	18	18	1755	213.25	稍小于边界值
Test17	18	18	18	1800	220	边界值
Test18	18	18	19	1825	225	稍大于边界值
Test19	18	19	18	1830	226	稍大于边界值
Test20	19	18	18	1845	229	稍大于边界值
Test21	48	48	48	4800	820	中间值
Test22	70	80	89	7775	1415	稍小于最大输出值
Test23	70	79	90	7770	1414	稍小于最大输出值
Test24	69	80	90	7755	1411	稍小于最大输出值
Test25	70	80	90	7800	1420	最大输出值

4.3.6 边界值分析的局限性

当被测程序含有多个独立变量,这些变量又受物理量的制约时,使用边界值分析测试方法比较合适,关键是‘独立’和‘物理量”。简单地看一下 NextDate 函数的边界值分析测试用例,就会发现这些测试用例是不充分的,例如,没有强调对 2 月和闰年的测试。问题的根源是,边界值分析假设变量是独立的,而 month、day 和 year 变量之间存在某些依赖关系。边界值分析测试用例通过使用物理量的边界导出变量极值,不考虑函数的性质,也不考虑变量的语法含义。即便如此,边界值分析测试也能捕获到一些月末和年末的缺陷。因此,边界值分析测试用例可以看做是初步的,这些测试用例的获得基本上不需要理解和想像。

物理量准则也很重要,如果变量引用了某个物理量,例如温度、压力、空气速度、负载等,物理边界就变得极为重要。一个有意思的例子是,菲尼克斯的国际机场在 1992 年 6 月 26 日被迫关闭,因为空气温度达到 120 ℉,以至于飞行员在起飞之前无法设置某一设备。该设备能够接受的最大空气温度是 120 ℉。另一个例子是,医疗分析系统使用步进电机确定要分析的样本在传送带上的位置,结果发现将传送带回送到开始单元格的过程,常常使机械手错过第一个单元格。

边界值分析不适用于逻辑变量和布尔型变量。例如,作为逻辑(相对于物理)变量的一个例子,很难想像 0000、0001、5000、9998 和 9999 这样的数字或电话号码会发现什么故障。尽管边界值分析测试很有用,但在实际运用中,并不如测试人员预想得那样令人满意。

基于函数(程序)输入定义域的测试方法,是所有测试方法中最基本的。这类测试方法都有一种假设,即输入变量是真正独立的,如果不能保证这种假设,则这类方法不能产生令人满意的测试用例(例如在 NextDate 函数中生成 1912 年 2 月 31 日)。这些方法都可以应用于程序的输出值域,就像在雇佣金问题中所做的一样。

4.4 决策表测试

在所有的黑盒测试方法中,基于决策表的测试是最严格、最具有逻辑性的测试方法。

4.4.1 决策表

在一些数据处理问题中,某些操作的实施依赖于多个逻辑条件的组合,即针对不同逻辑条件的组合值,分别执行不同的操作,决策表很适合于处理这类问题。自从 20 世纪 60 年代初以来,决策表一直被用来表示和分析复杂的逻辑关系,描述不同条件集合下采取行动的若干组合情况。

这里通过一个简单的例子,说明什么是决策表。

表 4 – 15 是一张名为‘阅读指南’的表单,表中列举了读者读书时可能遇到的 3 个问题,若读者的回答是肯定的(判定取真值),标以字母‘Y’;若回答是否定的(判定取假值),标以字母“N”。3 个判定条件,共有 8 种取值情况。该表还为读者提供了 4 条建议,但不需要每种情况都实施。要实施的建议在相应栏内标以‘√’,其他建议栏内则什么也不标。例如,表中的第 3 种情况,当读者已经疲劳,对内容又不感兴趣,并且还没读懂,这时建议读者去休息。这就是一张决策表。

表 4 – 15 阅读指南

选项 \ 规则		1	2	3	4	5	6	7	8
问题	你觉得疲倦吗？	Y	Y	Y	Y	N	N	N	N
	感兴趣吗？	Y	Y	N	N	Y	Y	N	N
	糊涂吗？	Y	N	Y	N	Y	N	Y	N
建议	重读					√			
	继续						√		
	跳到下一章							√	√
	休息	√	√	√	√				

决策表通常由 4 部分组成(如图 4 – 6 所示),它们分别是：

- 条件桩
- 条件项
- 动作桩
- 动作项

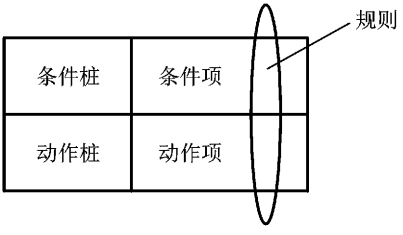


图 4 – 6 决策表组成

条件桩部分列出了问题的所有条件,除了某些问题对条件的先后次序有特定的要求外,通常在这里列出的条件其先后次序无关紧要。条件项部分针对条件桩给出的条件列出所有可能的取值。动作桩则给出了问题规定的可能采取的操作,这些操作的排列顺序一般没有什么约束,但为了便于阅读也可令其按适当的顺序排列。动作项和条件项紧密相关,指出在条件项的各组取值情况下应采取的动作。例如,在决策表 4 – 16 中,如果 c1、c2 和 c3 都为真,则采取动作 a1 和 a2。如果 c1 和 c2 都为真而 c3 为假,则采取动作 a1 和 a3。把任何一个条件组合的特定取值及相应要执行的动作称为一条规则,在决策表中贯穿条件项和动作项的一列就是一条规则。显然,决策表中列出多少组条件取值,就有多少条规则。

表 4 – 16 决策表

选项 \ 规则	规则 1	规则 2	规则 3,4	规则 5	规则 6	规则 7,8
条件 c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	–	T	F	–
动作 a1	√	√		√		
a2	√				√	
a3		√		√		
a4			√			√

实际使用决策表时,常常先将它简化,简化是以合并相似规则为目标的。若表中有两条或多条规则具有相同的动作,并且在条件项之间存在着极为相似的关系,便可以设法将其合并。例如,在决策表 4 – 16 中第 3、4 条规则其动作项一致,条件项中前 2 个条件取值一致,只是第 3 个条件取值不同,这一情况表明,前 2 个条件分别取真值和假值时,无论第 3 个条件取什么值,都要执行同一操作,即要执行的动作与第 3 个条件的取值无关。于是,便将这两个规则合并。合并后的第 3 条件项用符号‘ – ’表示与取值无关,称为‘ 无关条件 ’或‘ 不关心条件 ’。与此类似,具有相同动作的规则还可进一步合并,如图 4 – 7 所示。

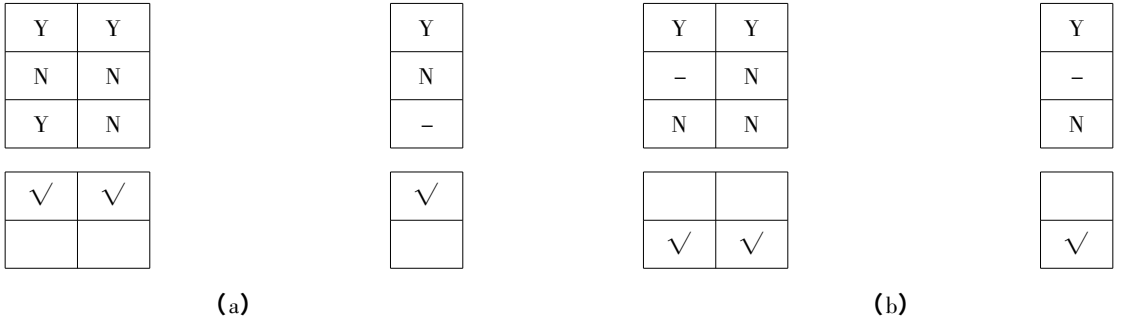


图 4 – 7 两条规则合并成一条

按上述合并规则,可将‘ 读书指南 ’决策表加以简化,简化后的判定表如表 4 – 17 所示。

表 4 – 17 简化后的阅读指南决策表

选项 \ 规则		1 – 4	5	6	7 – 8
问题	你觉得疲倦吗?	Y	N	N	N
	感兴趣吗?	–	Y	Y	N
	糊涂吗?	–	Y	N	–
建议	重读		√		
	继续			√	
	跳到下一章				√
	休息	√			

结合三角形问题给出构造决策表的 5 个步骤如下：

① 确定规则的个数。例如，三角形问题的决策表有 4 个条件，每个条件可以取两个值，故应有 $2^4 = 16$ 种规则。

② 列出所有的条件桩和动作桩。

③ 填入条件项。

④ 填入动作项，这样便可得到初始决策表。

⑤ 化简。合并相似规则后得到三角形问题的决策表 4 – 18。

表 4 – 18 三角形问题的决策表

选项 \ 规则	规则 1 ~ 8	规则 9	规则 10	规则 11	规则 12	规则 13	规则 14	规则 15	规则 16
条件：									
c1：a，b，c 构成一个三角形？	N	Y	Y	Y	Y	Y	Y	Y	Y
c2：a = b？	–	Y	Y	Y	Y	N	N	N	N
c3：a = c？	–	Y	Y	N	N	Y	Y	N	N
c4：b = c？	–	Y	N	Y	N	Y	N	Y	N
动作：									
a1：非三角形	√								
a2：一般三角形									√
a3：等腰三角形					√		√	√	
a4：等边三角形		√							
a5：不可能			√	√		√			

如果条件使用的是等价类，则决策表会有另一种典型的外观，如表 4 – 19 所示。决策表 4 – 19 来自 NextDate 问题，它引入了 month 变量相互排斥的可能性。由于一个月份就是一个等价类。不可能出现两项同时为真的规则，即 month 在 M1 中，当规则 1 为 T 时，规则 2 和规则 3 都不能为 T。这里，无关条件的实际含义是“不可能”。关于使用等价类的决策表，后面将结合 NextDate 函数进行讨论。

表 4 – 19 条件互斥的决策表

选项 \ 规则	规则 1	规则 2	规则 3
条件：			
c1：month 在 M1 中？	T	–	–
c2：month 在 M2 中？	–	T	–
c3：month 在 M3 中？	–	–	T
动作：			
a1			
a2			
a3			

4.4.2 决策表在黑盒测试中的应用

决策表最突出的优点是,它能把复杂的问题按各种可能的情况一一列举出来,简明而易于理解,同时可以避免遗漏。因此利用决策表可以设计出完整的测试用例集合。使用决策表设计测试用例,可以把条件解释为输入,把动作解释为输出。

1. 三角形问题的决策表测试用例设计

表4-18是一张三角形问题的决策表,还可以将条件(c1:a、b、c构成三角形?)扩展为三角形特性的3个不等式,c1:a<b+c、c2:b<a+c和c3:c<a+b。如果有一个不等式不成立,则3个整数就不能构成三角形,这样扩展后的决策表如表4-20所示。当然还可以进一步扩展,因为不等式不成立有两种方式:一条边等于另外两条边的和或严格大于另外两条边的和。

使用决策表4-20,可以得到11个测试用例如表4-21所示,其中3个测试用例检测不可能的情况,3个测试用例检测非三角形的情况,1个测试用例检测等边三角形的情况,1个测试用例检测一般三角形的情况,3个测试用例检测等腰三角形。如果扩展决策表显示两种违反三角形性质的方式时,可以再设计一些测试用例(一条边正好等于另外两条边的和)。做到这一点需要做一定的判断,否则规则会呈指数级增长,可能还会得到许多不可能的规则。

表4-20 扩展的三角形问题决策表

规则 / 选项	规则1~32	规则33~48	规则49~56	规则57	规则58	规则59	规则60	规则61	规则62	规则63	规则64
条件:											
c1:a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2:b<a+c?	–	F	T	T	T	T	T	T	T	T	T
c3:c<a+b?	–	–	F	T	T	T	T	T	T	T	T
c4:a=b?	–	–	–	T	T	T	T	F	F	F	F
c5:a=c?	–	–	–	T	T	F	F	T	T	F	F
c6:b=c?	–	–	–	T	F	T	F	T	F	T	F
动作:											
a1:非三角形	√	√	√								
a2:一般三角形											√
a3:等腰三角形							√		√	√	
a4:等边三角形				√							
a5:不可能					√	√		√			

表 4 – 21 三角形问题的决策表测试用例

测试用例	a	b	c	预期输出
Test1	4	1	2	非三角形
Test2	1	4	2	非三角形
Test3	1	2	4	非三角形
Test4	5	5	5	等边三角形
Test5	?	?	?	不可能
Test6	?	?	?	不可能
Test7	2	2	3	等腰三角形
Test8	?	?	?	不可能
Test9	2	3	2	等腰三角形
Test10	3	2	2	等腰三角形
Test11	3	4	5	一般三角形

2. NextDate 函数的决策表测试用例设计

NextDate 函数可以说明输入定义域中的依赖性问题,这使得它成为基于决策表测试的一个完美例子,因为决策表可以突出这种依赖关系。前面曾经介绍过 NextDate 函数的等价类划分。等价类划分的不足之处在于,从等价类中机械地选取输入值,可能会产生‘非常奇怪’的测试用例,例如找出 2000 年 4 月 31 日的下一天(见表 4 – 7)。问题的根源是等价类划分和边界值分析测试都假设变量是独立的。如果变量之间在输入定义域中存在某些逻辑依赖关系,那么这些依赖关系在机械地选取输入值时可能会丢失。决策表方法通过使用‘不可能动作’概念表示条件的不可能组合,来强调这种依赖关系。

为了产生给定日期的下一个日期 NextDate 函数能够使用的操作只有 5 种:day 变量和 month 变量的加 1 和复位操作,year 变量的加 1 操作。

如果将注意力集中到 NextDate 函数的日和月问题上,并仔细研究动作桩。可以在以下的等价类集合上建立决策表。

M1:{month:month 有 30 天}

M2:{month:month 有 31 天,12 月除外}

M3:{month:month 是 12 月}

M4:{month:month 是 2 月}

D1:{day:1 ≤ day ≤ 27}

D2:{day:day = 28}

D3:{day:day = 29}

D4:{day:day = 30}

D5:{day:day = 31}

Y1:{year:year 是闰年}

Y2：{year：year 不是闰年}

决策表如表 4－22 所示，共有 22 条规则。前 5 条规则（1～5）处理有 30 天的月份，接下来的两组规则（规则 6～10 和规则 11～15）处理有 31 天的月份，其中规则 6～10 处理 12 月之外的月份，规则 11～15 处理 12 月，不可能规则也在决策表中列出，比如规则 5（在有 30 天的月份中考虑 31 日）。一些资深测试人员可能会有疑问，这 10 条规则（6～15）中的 8 条都只是对 day 变量加 1，针对这个子功能真的需要 8 条单独的测试用例吗？可能不需要，但是通过决策表可以得到一些启发。最后的 7 条规则关注的是 2 月和闰年。

表 4－22 NextDate 函数的决策表

选项 \ 规则	1	2	3	4	5	6	7	8	9	10	11
条件：											
c1：month 在	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2	M3
c2：day 在	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1
c3：year 在	–	–	–	–	–	–	–	–	–	–	–
动作：											
a1：不可能					√						
a2：day 加 1	√	√	√			√	√	√	√		√
a3：day 复位				√						√	
a4：month 加 1				√						√	
a5：month 复位											
a6：year 加 1											

选项 \ 规则	12	13	14	15	16	17	18	19	20	21	22
条件：											
c1：month 在	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2：day 在	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3：year 在	–	–	–	–	–	Y1	Y2	Y1	Y2	–	–
动作：											
a1：不可能									√	√	√
a2：day 加 1	√	√	√		√	√					
a3：day 复位				√			√	√			
a4：month 加 1							√	√			
a5：month 复位				√							
a6：year 加 1				√							

使用决策表代数可以进一步简化这 22 条规则。如果决策表中两条规则的动作项相同,则一定至少有一个条件能够把这两条规则用不关心条件合并。例如,规则 1、2 和 3 都涉及有 30 天的月份的 day 类 D1、D2 和 D3,并且它们的动作项都是 day 加 1,则可以将规则 1、2 和 3 合并。类似地,有 31 天的月份的 day 类 D1、D2、D3 和 D4 也可以合并,2 月的 D4 和 D5 也可以合并。简化后的决策表如表 4 – 23 所示。

表 4 – 23 进一步简化后的 NextDate 函数决策表

选项 \ 规则	1 ~ 3	4	5	6 ~ 9	10	11 ~ 14	15	16	17	18	19	20	21,22
条件:													
c1 :month 在	M1	M1	M1	M2	M2	M3	M3	M4	M4	M4	M4	M4	M4
c2 :day 在	D1,D2,D3	D4	D5	D1,D2,D3,D4	D5	D1,D2,D3,D4	D5	D1	D2	D2	D3	D3	D4,D5
c3 :year 在	–	–	–	–	–	–	–	–	Y1	Y2	Y1	Y2	–
动作:													
a1 :不可能			√									√	√
a2 :day 加 1	√			√		√		√	√				
a3 :day 复位		√			√		√			√	√		
a4 :month 加 1		√			√					√	√		
a5 :month 复位							√						
a6 :year 加 1							√						

根据简化后的决策表 4 – 23,可设计测试用例如表 4 – 24 所示。

表 4 – 24 NextDate 函数的决策表测试用例

测试用例	month	day	year	预期输出
Test1 ~ 3	8	16	2001	17/8/2001
Test4	8	30	2004	1/9/2004
Test5	8	31	2001	不可能
Test6 ~ 9	1	16	2004	17/1/2004
Test10	1	31	2001	1/2/2001
Test11 ~ 14	12	16	2004	17/12/2004
Test15	12	31	2001	1/1/2002
Test16	2	16	2004	17/2/2001
Test17	2	28	2004	29/2/2004
Test18	2	28	2001	1/3/2001
Test19	2	29	2004	1/3/2004
Test20	2	29	2001	不可能
Test21 ~ 22	2	30	2004	不可能

3. 雇佣金问题的决策表测试用例

决策表测试不太适合于雇佣金问题。因为在雇佣金问题中只有很少的决策逻辑。由于雇佣金问题等价类中的变量是真正独立的,决策表中没有不可能规则,因此,得到的测试用例与等价类分析测试用例一样。

4.4.3 决策表测试的指导方针

与其他测试方法一样,基于决策表的测试可能对于某些应用程序(例如 NextDate 函数)很有效,但对另外一些应用程序(例如雇佣金问题)就不值得费这么大的精力。基于决策表测试适用于要产生大量决策的情况(例如三角形问题),或在输入变量之间存在重要的逻辑关系的情况(例如 NextDate 函数)。

① 决策表测试方法适用于具有以下特征的应用程序:

- if – then – else 逻辑突出。
- 输入变量之间存在逻辑关系。
- 涉及输入变量子集的计算。
- 输入与输出之间存在因果关系。

② 适合于使用决策表设计测试用例的情况有:

- 规格说明以决策表形式给出,或是很容易转换成决策表。
- 条件的排列顺序不会也不应影响执行的操作。
- 规则的排列顺序不会也不应影响执行的操作。
- 当某一规则的条件已经满足,并确定要执行的操作后,不必检验别的规则。
- 如果某一规则要执行多个操作,这些操作的执行顺序无关紧要。

给出这些情况的目的是为了说明操作的执行应完全依赖于条件的组合。其实对于某些不满足这几条的决策表,同样可以用来设计测试用例,只不过需增加一些其他的测试用例罢了。

③ 决策表规模较大,有 n 个条件的有限条目决策表(每个条件取真、假值)有 2^n 个规则。现在已有多种方法可以解决这个问题——扩展条目决策表(条件使用等价类)、代数简化表,将大表‘分解’为小表等方法。

④ 与其他方法一样,迭代也比较有效。第一次识别的条件或动作可能不那么令人满意,把第一次得到的结果作为铺路石,逐渐改进,直到得到满意的决策表为止。

4.5 其他黑盒测试方法

4.5.1 因果图

等价类划分和边界值分析方法,着重考虑输入条件,不考虑输入条件的各种组合,也不考虑各个输入条件之间的相互制约关系。如果在测试时必须考虑输入条件的各种组合,可能的组合数将是一个天文数字,因此必须考虑使用一种适合于描述多种条件的组合,产生多个相应动作的测试方法,这就需要因果图。因果图方法能够帮助测试人员按照一定的步骤,高效率地开发测试用例,以检测程序输入条件的各种组合情况。它是将自然语言规格说明转化成形式语言规格说

明的一种严格的方法,可以指出规格说明中存在的不完整性和二义性。

下面先介绍因果图,因果图中使用了简单的逻辑符号,以直线连接左右结点。左结点表示输入状态(或称原因),右结点表示输出状态(或称结果)。因果图中用 4 种符号分别表示规格说明中的 4 种因果关系。图 4-8 给出了因果图中常用的 4 种符号所代表的因果关系。

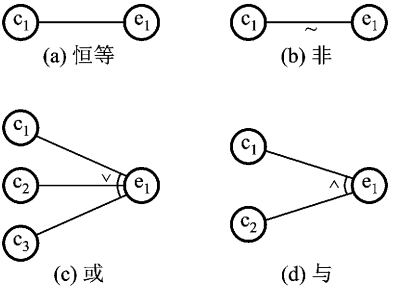


图 4-8 因果图的基本符号

图中 c_i 表示原因,通常位于图的左部;e_i 表示结果,位于图的右部。c_i 和 e_i 都可以取值 0 或 1,0 表示某状态不出现,1 表示某状态出现。

- 恒等:若 c_1 是 1,则 e_1 也是 1;否则 e_1 为 0。
- 非:若 c_1 是 1,则 e_1 是 0;否则 e_1 为 1。
- 或:若 c_1 或 c_2 或 c_3 是 1,则 e_1 是 1;否则 e_1 为 0。“或”可以有任意个输入。
- 与:若 c_1 和 c_2 都是 1,则 e_1 为 1;否则 e_1 为 0。“与”也可以有任意个输入。

在实际问题中,输入状态相互之间还可能存在某些依赖关系,称之为“约束”。比如,某些输入条件本身不可能同时出现。输出状态之间也往往存在约束。在因果图中,用特定的符号标明这些约束(见图 4-9)。

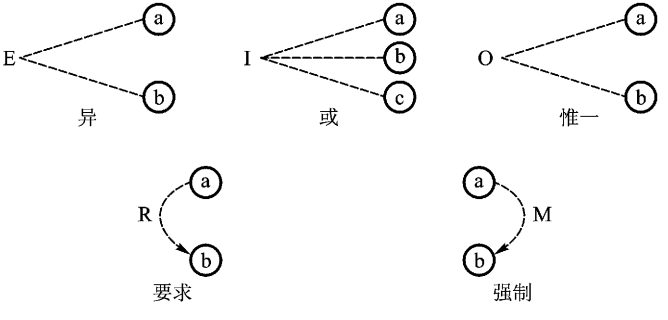


图 4-9 约束符号

对于输入条件的约束有以下 4 种。

- E 约束(异):a 和 b 中最多有一个可能为 1,即 a 和 b 不能同时为 1。
- I 约束(或):a、b 和 c 中至少有一个必须是 1,即 a、b 和 c 不能同时为 0。
- O 约束(惟一):a 和 b 必须有一个且仅有 1 个为 1。
- R 约束(要求):a 是 1 时,b 必须是 1,即当 a 是 1 时,b 不能是 0。

输出条件的约束只有 M 约束。

• M 约束(强制):若结果 a 是 1,则结果 b 强制为 0。

因果图方法最终生成决策表。利用因果图导出测试用例需要经过以下几个步骤:

① 分析程序规格说明中哪些是原因,哪些是结果。原因常常是输入条件或输入条件的等价类,结果则是输出条件。

② 分析程序规格说明中语义的内容,找出原因与结果之间,原因与原因之间的对应关系,并将其表示成连接各个原因与各个结果的"因果图"。

③ 由于语法或环境的限制,有些原因与原因之间,原因与结果之间的组合情况不可能出现。为表明这些特定的情况,在因果图上使用一些记号标明约束或限制条件。

④ 把因果图转换成决策表。

⑤ 根据决策表中每一列设计测试用例。

下面以一个简单的例子,说明因果图方法的步骤。

某软件规格说明要求:第一个字符必须是#或 *,第二个字符必须是一个数字,在此情况下进行文件的修改。如果第一个字符不是#或 *,则给出信息 N;如果第二个字符不是数字,则给出信息 M。

在分析以上的要求以后,可以明确地把原因和结果分开。

原因:

c_1——第一个字符是#。

c_2——第一个字符是 *。

c_3——第二个字符是一数字。

结果:

e_1——给出信息 N。

e_2——修改文件。

e_3——给出信息 M。

将原因和结果用上述的逻辑符号联接起来,可以得到如图 4-10 所示的因果图。图中左边表示原因,右边表示结果,编号为 10 的中间结点是导出结果的进一步原因。

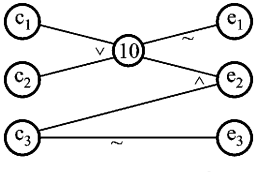


图 4-10 因果图表示

考虑到原因 c_1 和 c_2 不可能同时为 1,即第一个字符不可能既是#又是 *,在因果图上可对其施加 E 约束,这样便得到了具有约束的因果图,如图 4-11 所示。根据因果图可以建立如表 4-25所示的决策。

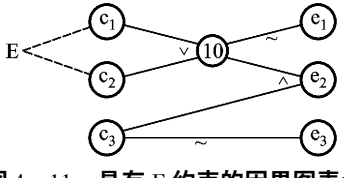


图 4-11 具有 E 约束的因果图表示

表 4-25 根据因果图建立的决策表

规则 / 选项	1	2	3	4	5	6	7	8
条件：								
c_1	1	1	1	1	0	0	0	0
c_2	1	1	0	0	1	1	0	0
c_3	1	0	1	0	1	0	1	0
10			1	1	1	1	0	0
动作：								
e_1							√	√
e_2			√		√			
e_3				√		√		√
不可能	√	√						
测试用例			#3	#A	*6	*B	A1	GY

注意，表中 8 种情况的最左面两列，原因 c_1 和 c_2 同时为 1，这是不可能的，故应排除这两种情况。根据该表，可设计出 6 个测试用例，见表最后一行。

以上只是因果图应用的一个简单例子，但不要以为因果图是多余的。事实上，在较为复杂的问题中，这个方法常常十分有效，它能有效地帮助我们检查输入条件组合，设计出非冗余、高效的测试用例。当然，如果开发项目在设计阶段就采用了决策表，就不必再画因果图，可以直接利用决策表设计测试用例。

4.5.2 特殊值测试

特殊值测试是最直观、运用得最广泛的一种测试方法。当测试人员应用其领域知识使用类似程序的测试经验等信息开发测试用例时，常常使用特殊值测试。这种方法不使用测试策略，只根据‘最佳工程判断’来设计测试用例。因此，特殊值测试特别依赖测试人员的能力。

特殊值测试非常有用。如果为 NextDate 函数定义特殊值测试用例，多个测试用例可能会涉及 2 月 28 日、2 月 29 日和闰年。尽管特殊值测试具有高度的主观性，但是所产生的测试用例集合，常常比用其他方法生成的测试集合具有更高的测试效率，更能有效地发现软件故障。

4.5.3 故障猜测法

人们也可以靠经验和直觉猜测程序中可能存在的各种软件故障，从而有针对性地编写检查这些故障的测试用例。这就是故障猜测法，它是一种很特别的方法。

故障猜测法的基本思路是列出程序中所有可能出现的故障或容易发生故障的情况，然后根据它们开发测试用例。比如以前遇到的最容易出错的情况是什么？故障的历史可能提供一些答

案,过去出错的地方很可能以后还会出错。在介绍单元测试时,曾给出许多在模块中常见的故障,这些是单元测试经验的总结。此外,对于在程序中容易发生故障的情况,也有一些经验总结。例如,输入数据为0或输出数据为0是容易发生故障的情形,因此可选择使输入数据为0或使输出数据为0的测试用例。又如,输入表格为空或输入表格只有一行,也是容易发生故障的情况,可选择表示这种情况的例子作为测试用例。再如,针对一个排序程序可以输入空值(没有数据)、输入一个数据、让所有的输入数据都相等、让所有输入数据有序排列、让所有输入数据逆序排列等,进行故障推测。

Myers从事的一项研究表明程序中剩余故障的概率与已经发现的故障成比例。仅仅这一条,就为高效率的故障猜测提供了空间。

4.6 黑盒测试效率

上面讨论了几种典型的黑盒测试方法,这些测试方法的共同特点是,它们都把被测程序看做是一个打不开的黑盒,只知道输入到输出的映射关系,根据软件规格说明设计测试用例。在等价类分析测试中,通过等价类划分来减少测试用例的绝对数量。边界值分析方法则通过分析输入变量的边界值域设计测试用例。在基于决策表的测试中,通过分析被测程序的逻辑依赖关系,构造决策表,进而设计测试用例。随后,简单介绍了因果图测试方法,特殊值测试以及故障猜测方法等。那么,哪种测试方法最好?如何有效地选择测试方法?下面从测试工作量、测试效率两方面来讨论,它们是进行有效测试的关键。

1. 测试工作量

主要以边界值分析、等价类划分和决策表测试方法来讨论它们的测试工作量,即生成测试用例的数量与开发这些测试用例所需的工作量。图4-12给出了这3种测试方法的测试用例数量的曲线。

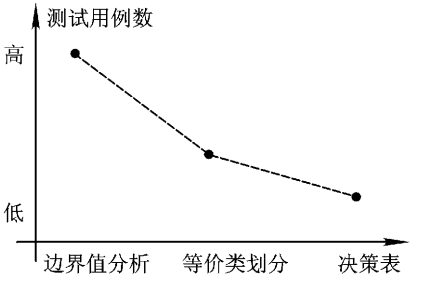


图4-12 每种测试方法的测试用例数量

边界值分析测试方法不考虑数据或逻辑依赖关系,它机械地根据各边界生成测试用例生成的测试用例最多。等价类划分测试方法则关注数据依赖关系和函数本身,需要借助于判断和技巧,考虑如何划分等价类,随后也是机械地从等价类中选取测试输入,生成测试用例。决策表技术最精细,它要求测试人员既要考虑数据,又要考虑逻辑依赖关系。当然,也许要经过几次尝试才能得到令人满意的决策表,但是如果有了一个良好的条件集合,所得到的测试用例就是完备

的,在一定意义上讲也是最少的。图 4 – 13 则说明了由每种方法设计测试用例的工作量曲线。由此可以看出,决策表测试用例生成所需的工作量最大。

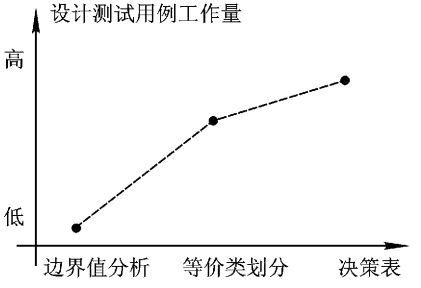


图 4 – 13 每种方法设计测试用例的工作量趋势

边界值分析测试方法使用简单,但会生成大量测试用例,机器执行时间很长。如果将精力投入到更精细的测试方法,如决策表方法,则测试用例生成花费了大量的时间,但生成的测试用例数少,机器执行时间短。这一点很重要,因为一般测试用例都要执行多次。测试方法研究的目的就是在开发测试用例工作量和测试用例执行工作量之间做一个令人满意的折中。

2. 测试效率

如果仔细研究这 3 个例子的测试用例集合,可以发现黑盒测试的基本局限是,遗漏了某些未测试的功能和冗余测试。例如 NextDate 问题,健壮性边界值测试产生 19 个测试用例(表 4 – 13)。仔细研究这些测试用例,就会发现结果并不令人满意 通过测试用例 1 ~ 7 预期能够得到什么结论?这些测试用例通过 5 个不同年份中的 6 月 15 日检查 NextDate 函数。年对日历的这个部分没有关系,因此这种测试用例有一个就足够的。再进一步研究,发现只有 1 个测试用例对 2 月进行测试,但是没有涉及 28 或 29 日,并且与闰年也没有联系。因此不仅有大量的冗余,而且围绕 2 月底和闰年,测试还存在严重漏洞。

健壮等价类测试有 10 个测试用例(表 4 – 7),其中 2 个是不可能的。这些不可能测试用例的根源,是由于等价类之间的独立性假设。这些测试用例中大多数用于处理非 2 月的 28 日问题,因此没有多大意思。决策表产生了 13 个测试用例(表 4 – 24),它们测试了健壮等价类测试用例遗漏的可能性,可以说这些测试用例在某种意义上是完备的。所有这些分析都支持两点结论 功能性测试有漏洞,使用更精细的测试手段可以减少这些漏洞。

一般来说,更精细的方法有助于识别漏洞,但是却不能保证什么。因为可以为某个程序开发出优秀的健壮等价类,然后又构造出很差的决策表。

3. 测试有效性

关于测试用例集合,真正想知道的是它们的测试效果如何,即一组测试用例找出程序中故障的效率如何。但是,解释测试有效性是很困难的,因为不可能知道程序中的所有故障,因此也就不可能知道给定方法所产生的测试用例是否能够发现这些故障,所能够做的,只是根据不同类型的故障,选择最有可能发现这种缺陷的测试方法(包括白盒测试)。根据最有可能出现的故障种类,分析得到可提高测试有效性的实用方法。通过跟踪所开发软件中的故障的种类和密度,也可以改进测试方法。这需要测试经验和技巧。

以下这个小故事对测试人员很有启示:有一个醉汉在路灯下的人行道上爬行,当警察问他在干什么时,他说他在找汽车钥匙。“你是在这里丢的吗?”警察问,“不,我在停车场丢的,但是这里的光线更亮些。”作为软件测试人员来说,测试不大可能存在的故障是没有意义的,只有很好地了解最有可能发生的故障种类,选择最有可能发现这类缺陷的测试方法,才有可能使测试最有效。

有些时候,人们可能对普遍存在的故障没有任何感觉。这时最好的办法是利用程序的已知属性,选择处理这些属性的方法。在选择黑盒测试方法时一些经常用到的属性有:

- 变量表示物理量还是逻辑量。
- 在变量之间是否存在依赖关系。
- 是否有大量的例外处理。

下面列出一些黑盒测试方法选取的初步“专家系统”:

- 如果变量引用的是物理量,可采用边界值分析测试和等价类测试。
- 如果变量是独立的,可采用边界值分析测试和等价类测试。
- 如果变量不是独立的,可采用决策表测试。
- 如果可保证是单故障假设,可采用边界值分析和健壮性测试。
- 如果程序包含大量例外处理,可采用健壮性测试和决策表测试。
- 如果变量引用的是逻辑量,可采用等价类测试用例和决策表测试。

小结

黑盒测试方法的共同特点是将被测程序看做一个打不开的黑盒,只根据软件规格说明设计测试用例。常用的黑盒测试方法有等价类划分、边界值分析、决策表法等。

等价类划分把程序的输入域划分成若干个互不相交的等价类,其目的是要减少测试用例的绝对数量。在进行等价类划分测试时,不仅要考虑有效等价类,还应考虑无效等价类。

边界值分析方法则通过分析输入变量的边界值域设计测试用例。实践表明,在设计测试用例时,对边界及次边界附近的处理必须给予足够的重视。为检验边界附近的处理,专门设计测试用例,常常可以取得良好的测试效果。在基于决策表的测试中,通过分析被测程序的逻辑依赖关系,构造决策表,进而设计测试用例。在所有的黑盒测试方法中,基于决策表的测试是最具有逻辑性的测试方法,但所需的测试工作量较大。

各种黑盒测试方法各有所长,应针对软件开发项目的具体要求,选择适当的测试方法,设计高效的测试用例,有效地将软件中隐藏的故障揭露出来。

第4章习题

1. 黑盒测试的最大问题是什么?
2. 为什么了解代码的执行方式会影响测试的方式或内容?
3. 健壮等价类测试与标准等价类测试的主要区别是什么?
4. 启动 Word 程序并从 File 菜单中选择 Print 命令,打开打印对话框,左下角显示的 Print Range(打印区域)

存在什么样的边界条件？

5. 对三角形问题的一种常见补充是检查直角三角形。如果满足毕达哥拉斯（Pythagorean）关系（$c^2 = a^2 + b^2$），则三条边构成直角三角形。试针对包含了直角三角形的扩展三角形问题来设计标准等价类测试用例。

6. 试为三角形问题中的直角三角形开发一个决策表和相应的测试用例。注意，会有等腰直角三角形。

第5章　白盒测试

白盒测试方法的突出特点是基于被测程序的源代码，而不是软件的规格说明。和其他软件测试技术相比，白盒测试方法更容易发现软件故障。本章将介绍几种常见的白盒测试方法，如逻辑覆盖、数据流测试、域测试、符号测试、路径分析、程序变异以及程序插装技术等，其中多数方法比较成熟，也有较高的实用价值，个别方法存在一定的局限性。

本章重点：

- 程序控制流图
- 逻辑覆盖
- 路径分析
- 数据流测试
- 域测试策略
- 符号测试
- 程序变异
- 程序插装

5.1　程序控制流图

程序控制流图是白盒测试的主要依据。对于一个程序，其程序控制流图 G = (V,E,I,O) 是一个有向图，其中 V 是结点的集合，E 是边的集合，I 是惟一的源结点(入口结点)，而 O 是惟一的汇结点(出口结点)。在控制流图中：

- 结点表示语句或语句片段，以标有编号的圆圈表示。
- 边表示语句或语句片段间可能的控制流向。
- I,O 相应于程序的开始语句和结束语句。

如果 i 和 j 是程序控制流图中的结点，从结点 i 到结点 j 存在一条边，当且仅当对应结点 j 的语句或语句片段可以在对应结点 i 的语句或语句片段之后立即执行。本章讨论中，将语句或语句片段简称为语句。

根据给定程序构造其程序控制流图非常容易。一个问题是如何处理不可执行语句，例如注释和数据说明语句，最简单的解决办法是不考虑这些语句。以前面三角形问题的类 C 语言实现为例，针对下面的三角形问题应用类 C 语言来实现的程序控制流图如图 5 - 1 所示。

```
/* PROGRAM TRIANGLE */
mian ()
 {
1   int a,b,c;
2   boolean IsTraingle
```

```
3   sacnf("Enter 3 integer which is sides of a triangle. %d%d%d",a,b,c);
4   printf("Side a is %d", a);
5   printf("Side b is %d", b);
6   printf("Side c is %d",c);
7   if ((a<b+c) AND (b<a+c) AND (c<a+b))
8     then IsTraingle=True;
9     else IsTraingle=False;
10  if (IsTraingle)
11      then if ((a=b) AND (b=c))
12                then printf("Equilateral")
13                     else if (a≠b) AND (a≠c) AND (b≠c)
14                          then printf("Scalene")
15                               else printf("Isosceles")
16      else printf("Not a Triangle")
}
```

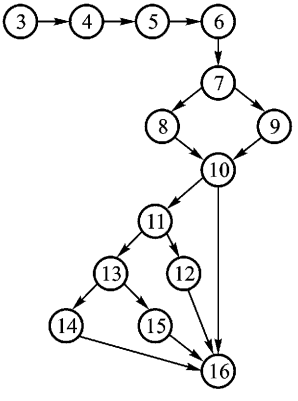


图 5-1 三角形程序的程序控制流图

在图 5-1 中,结点 3 到结点 7 是一个序列,结点 7 到结点 10 是一个 if-then-else 结构,结点 11 到结点 16 是一个嵌套的 if-then-else 结构。结点 3 和结点 16 是程序源结点和汇结点,对应于单入口、单出口准则。该程序没有循环,因此控制流图是一个有向非循环图。

程序路径是一个重要的概念,其定义如下:

若 G 中的结点序列 $p=<v_1,v_2,\ldots,v_k>$,满足对于所有的 j,有$(v_{i+j},v_{i+j+1})\in E$,$(0\leqslant j<k-i)$,则称 p 为程序的一条子路经。若 $v_1=v_k$,则称 P 为程序的一条回路(环路)。若子路径 p 从源结点 I 到汇结点 O,则称 p 为程序的一条路径。例如 $P=<3,4,5,6,7,8,10,11,12,16>$ 是一条程序路径。

程序控制流图的重要性在于,程序的执行对应于从源结点到汇结点的路径。检验程序从入口开始,执行过程中经历的各个语句,直到出口,是白盒测试最为典型的问题。

5.2 逻辑覆盖

逻辑覆盖以程序内部的逻辑结构为基础设计测试用例,要求对被测程序的逻辑结构有清楚的了解,甚至要掌握源程序的所有细节。由于覆盖测试的目标不同,逻辑覆盖又可分为:语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖及路径覆盖。

为说明几种逻辑覆盖测试方法之间的不同,可以结合下面的一小段程序来讨论。

```
begin
if((age>25) AND(sex=M)) then comm=comm+150;
if(age>=50 OR (comm>2000.0)) then comm=comm-200;
end;
```

其中,AND、OR 是逻辑运算符,3 个输入参数是年龄 age(整数)、性别 sex(男或女)和雇佣金 comm(实数)。图 5-2 给出了它的程序控制流程图,a、b、c、d 和 e 为控制流上的若干程序点。

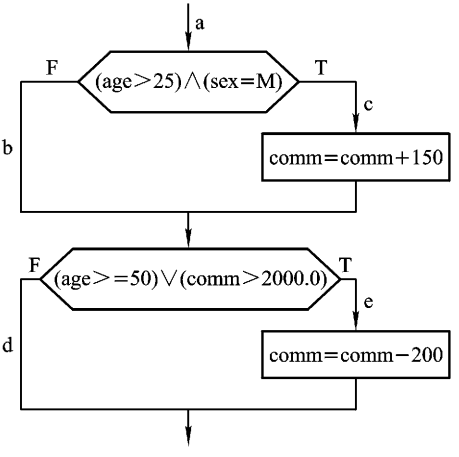


图 5-2 被测程序段流程图

1. 语句覆盖

语句覆盖要求设计若干个测试用例,运行被测程序,使程序中的每个可执行语句至少被执行一次。这里所谓'若干",自然是越少越好。

在上述程序段中,如果选择:

age=50,sex=M,comm=2500 Test1

作为测试用例,则程序按路径 ace 执行。这样该程序段的 4 个语句都被执行,从而做到了语句覆盖。如果选择:

age=50,sex=F,comm=2500 Test2

程序则按路径 abe 执行,没有达到语句覆盖。

从程序中每个语句都得到执行这一点来看,语句覆盖似乎能够比较全面地检验被测程序的每一个语句,但语句覆盖是很弱的逻辑覆盖准则。假如这一程序段中两个判断的逻辑运算符有问题,例如,第一个判断的运算符 AND 错写成运算符 OR 或是第二个判断中的运算符 OR 错写成

了运算符 AND,这时仍使用上述的测试用例 Test1,程序仍将按路径 ace 执行,但不能发现判断中逻辑运算的错误。

此外,还可以很容易地找出已经满足了语句覆盖,却仍然存在错误的例子。例如,若有一程序段:

```
...
if (age > =50)
   then comm = comm + 1500
...
```

如果错写成:

```
...
if(age >50)
   then comm = comm + 1500
...
```

假如给出的测试数据能够确保该程序段执行时 age 的值大于 50,那么 comm 被赋予新的值,虽然做到了语句覆盖,但却没有发现其中的故障。

实际上,和后面将介绍的其他几种逻辑覆盖相比较,语句覆盖是比较弱的覆盖准则。做到了语句覆盖可能给人们一种心理的满足,以为每个语句都执行过,似乎可以放心了,其实这仍然不是十分可靠的。语句覆盖在测试被测程序时,除去对检查不可执行语句有一定作用外,并不能排除被测程序中存在故障的风险。必须意识到,被测程序并非是语句的无序堆积,语句之间存在着许多有机的联系。

2. 判定覆盖

判定覆盖要求设计若干测试用例,运行被测程序,使得程序中每个判断的取真分支和取假分支至少执行一次,即判断的真假值均要被检测。判定覆盖又称为分支覆盖。

仍以上述程序段为例,若选用一组测试用例:

age = 50,sex = M,comm = 2500　　Test1

age = 20,sex = M,comm = 1500　　Test3

则分别执行路径 ace 和 abd,使两个判断的 4 个分支 b,c 和 d,e 分别得到检测。

也可以选用另外一组测试用例:

age = 40,sex = M,comm = 1500　　Test4

age = 50,sex = F,comm = 1900　　Test5

则分别路径 acd 及 abe,同样也可覆盖 4 个分支。

上述两组测试用例不仅满足了判定覆盖,同时还满足语句覆盖。从这一点看似乎判定覆盖比语句覆盖更强一些,但是,如果将此程序段中的第 2 个判断条件 comm > 2000 错写成 comm < 2000,使用上述测试用例 Test5,照样能按原路径执行(abe),不影响结果。这说明,只做到判定覆盖仍无法发现判断内部条件的错误。因此,需要有更强的逻辑覆盖准则去检验判断语句内的条件。

上面只考虑了两个出口的判断,判定覆盖准则还可以扩充到多出口判断(如 CASE 语句)的情况。

3. 条件覆盖

条件覆盖要求设计若干测试用例,执行被测程序,使得程序中每个判断的每个条件的可能取值至少被执行一次。

在上述程序段中,第一个判断应考虑到:

age > 25 取真值,记为 T1

age > 25 取假值,即 age ≤ 25,记为$\overline{T1}$

sex = M 取真值,记为 T2

sex = M 取假值,即 sex = F,记为$\overline{T2}$

第 2 个判断应考虑到:

age ≥ 50 取真值,记为 T3

age ≥ 50 取假值,即 age < 50,记为$\overline{T3}$

comm > 2000 取真值,记为 T4

comm > 2000 取假值, 记为$\overline{T4}$

前面设计的 3 个测试用例 Test1、Test3 和 Test5,执行该程序段所走路径及覆盖条件如表5 – 1 所示。

表 5 – 1 条件覆盖情况

测试用例	age	sex	comm	所走路径			覆盖条件			
Test1	50	M	2500.0	a	c	e	T1	T2	T3	T4
Test3	20	M	1500.0	a	b	d	$\overline{T1}$	T2	$\overline{T3}$	$\overline{T4}$
Test5	50	F	1900.0	a	b	e	T1	$\overline{T2}$	T3	$\overline{T4}$

从表 5 – 1 可以看出,3 个测试用例覆盖了 4 个条件的 8 种情况,即 T1、$\overline{T1}$、T2、$\overline{T2}$、T3、$\overline{T3}$、T4 和$\overline{T4}$。

进一步分析上表,覆盖了 4 个条件的 8 种情况的同时,也覆盖了两个判断的 4 个分支 b、c、d 和 e。是否可以这样说,做到了条件覆盖,也就必然实现了判定覆盖呢?分析下面的测试用例 Test6 和 Test7,执行程序段的覆盖情况如表 5 – 2 所示。

表 5 – 2 另一条件覆盖情况

测试用例	age	sex	comm	所走路径			覆盖分支		覆盖条件			
Test6	20	M	2100.0	a	b	e	b	e	$\overline{T1}$	T2	$\overline{T3}$	T4
Test7	50	F	1500.0	a	b	e	b	e	T1	$\overline{T2}$	T3	$\overline{T4}$

从表 5 – 2 可以看出,覆盖了条件的测试用例不一定覆盖了分支。事实上,它只覆盖了 4 个分支中的两个,即 b 和 e。为解决这一矛盾,需要对条件和分支进行兼顾测试。

4. 判定/条件覆盖

判定/条件覆盖要求设计足够的测试用例,执行被测程序,使得判断中每个条件的所有可能取值至少被执行一次,同时每个判断的所有可能判断结果也至少被执行一次。

在上述程序段中,两个判断各包含两个条件,这4个条件在两个判断中可能有8种组合,它们是:

① age > 25,sex = M 记为 T1,T2

② age > 25,sex = F 记为 T1,$\overline{T2}$

③ age ≤ 25,sex = M 记为$\overline{T1}$,T2

④ age ≤ 25,sex = F 记为$\overline{T1}$,$\overline{T2}$

⑤ age ≥ 50,comm > 2000.0 记为 T3,T4

⑥ age ≥ 50,comm ≤ 2000.0 记为 T3,$\overline{T4}$

⑦ age < 50, comm > 2000.0 记为$\overline{T3}$,T4

⑧ age < 50, comm ≤ 2000.0 记为$\overline{T3}$,$\overline{T4}$

可以设计4个测试用例,覆盖上述8种条件组合,见表5-3所示。

表5-3 判定/条件覆盖情况

测试用例	age	sex	comm	所走路径			覆盖组合		覆盖条件			
Test1	50	M	2500	a	c	e	①	⑤	T1	T2	T3	T4
Test6	20	M	2100.0	a	b	e	③	⑦	$\overline{T1}$	T2	$\overline{T3}$	T4
Test7	50	F	1500.0	a	b	e	②	⑥	T1	$\overline{T2}$	T3	$\overline{T4}$
Test8	20	F	1500	a	b	d	④	⑧	$\overline{T1}$	$\overline{T2}$	$\overline{T3}$	$\overline{T4}$

这一程序段共有4条路径即abd、abe、acd和ace。以上4个测试用例固然覆盖了所有的条件组合,同时也覆盖了4个分支,但只覆盖了3条路径,漏掉了路径acd。而路径能否被全面覆盖在软件测试中是一个重要的问题,因为程序要取得正确的结果,就必须消除遇到的各种障碍,沿着特定的路径顺利执行。如果程序中的每一条路径都得到考验,才能说程序受到了全面检验。

5. 路径覆盖

路径覆盖要求设计足够多的测试用例,覆盖程序中所有可能的路径。

针对上述程序段中的4条可能路径:

P1: ace

P2: abd

P3: acd

P4: abe

4个测试用例Test 1、Test 3、Test 4和Test 7,可以分别覆盖这4条路径。当然,也可以设计出其他的测试用例,覆盖这4条路径。

这里所用的程序段非常简短,只有4条路径。但在实际问题中,一个不太复杂的程序,其路径数都可能是一个庞大的数字,以致要在测试中覆盖所有的路径是不可能实现的。为解决这一难题,只得把覆盖的路径数压缩到一定限度内,例如,程序中的循环体只执行了一次。

即使对于路径数有限的程序做到了路径覆盖,也不能保证被测程序的正确性。例如,在上述语句覆盖一段中最后给出的故障也不是路径覆盖可以发现的。

由此看出,各种结构测试方法都不能保证程序的正确性。但是,测试的目的并不是要证明程序的正确性,而是要尽可能找出程序中隐藏的故障。事实上,并不存在一种十全十美的测试方

法,能够发现所有的软件故障。想要撒下几网就把湖中的鱼全都捕上来是不可能的。

5.3 路径分析

从上节的讨论中可以看出,对于比较简单的小程序来说,实现路径覆盖是可能的,但如果程序中出现了多个判断和多个循环,可能的路径数目将会急剧增长,以致实现路径覆盖是不可能的。实际上可以做到的只是有选择地测试程序中某些有代表性的路径。独立路径选择和 Z 路径覆盖是两种常用的路径覆盖方法。

本节将从程序路径表示入手,讨论路径数目的计算方法,从而看出路径数是如何随程序复杂度增加而增长的。

5.3.1 程序路径表示

进行路径分析,首先要解决的是路径的表示问题,下面给出弧序列表示、结点序列表示以及路径表达式这 3 种路径表示方法。这些方法应该比较直观、形象和便于人们理解的,同时还必须易于计算机处理。

1. 路径的弧序列表示及结点序列表示

程序结构常用程序控制流图表示。有了控制流图,给出程序路径的表示就容易多了。例如图 5 – 3 中的结点 I 表示程序入口,结点 O 表示程序出口,从 G 到 A 是一个循环,则从入口到出口的路径有许多条,在此列举 4 条路径,分别以弧序列和结点序列表示(见表 5 – 4)。

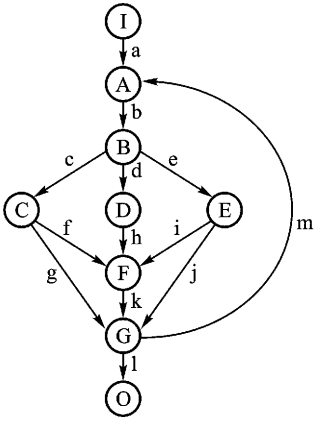


图 5 – 3 数以万亿计的路径

表 5 – 4 路径的弧序列和结点序列表示

弧序列表示	结点序列表示	进入循环次数
abdhkl	I – A – B – D – F – G – O	0
abcfkl	I – A – B – C – F – G – O	0
abcfkmbcfkl	I – A – B – C – F – G – A – B – C – F – G – O	1
abcfkmbeikl	I – A – B – C – F – G – A – B – E – F – G – O	1

2. 路径表达式

对于程序的某一条路径,可采用上述的弧序列表示或结点序列表示。是否可以找到一种能够给出程序中所有路径的、更加概括的表示方法呢?路径表达式可以满足这一要求。

路径表达式作为一种表达式,其运算对象指的是控制流图中的边(弧),此外还引入了两个运算 乘和加。

① 弧 a 与弧 b 相加,其和 a + b 表示两弧是或的关系。如图 5 - 4(a)中结点 2 至结点 3 间的两条弧,它们是并行关系,这时运算符' + '不可以省略。

② 弧 a 和弧 b 相乘,所得的乘积为 ab,它表示先沿弧 a,再沿弧 b 所经历的路段。注意,这里也同代数式一样,事实上省略了 a 和 b 之间的乘法运算符。其实这就是前面给出的弧序列表示,形式上没有什么差别,不过这时应该认为,各弧之间是一种相乘的关系,例如图 5 - 4(a)中 eacf 是 4 条弧的乘积,它表示沿着 e、a、c 和 f 的顺序所经历的路段,如图 5 - 4(b)所示。

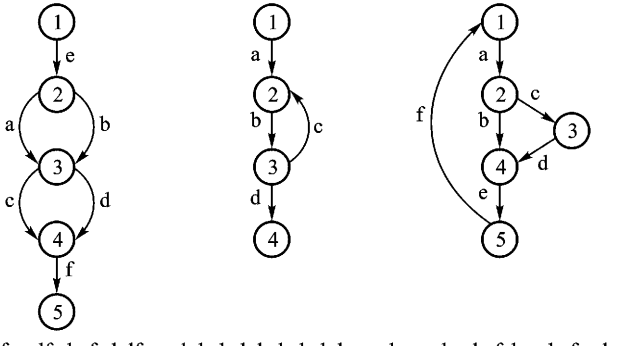


图 5 - 4 两个简单程序的控制流

在图 5 - 4(a)所表示的控制流图中,共有 4 条路径,它们分别是:eacf、eadf、ebcf 和 ebdf。这 4 条路径是并行(或)的关系,完全可以用加运算联接它们,从而得到完整的路径表达式:

eacf + eadf + ebcf + ebdf

另一方面,从图 5 - 4(a)中还可以看出,弧 a 和弧 b 是并行的,弧 c 和弧 d 也是并行的,它们的头尾又有弧 e 和弧 f 相连。按上述两种运算的定义可写出路径表达式:

e(a + b)(c + d)f

不难发现,这一表达式正是前面 4 项之和的因子提出形式。其实这并不是一个特殊情况的巧合,而是在路径表达式中普遍适用的分配律。

图 5 - 4(b)给出了一个具有循环的控制流图,它的路径随执行循环体的次数不同而有所不同,比如 abd、abcbd、abcbcbd... 分别是执行一次、2 次、3 次循环体的路径,其路径表达式可以写成:

abd + abcbd + abcbcbd + ...

进一步可以化简为:

ab(1 + cb + cbcb + ...)d 或 $ab(1 + cb + (cb)^2 + ...)d$

事实上,路径表达式中的运算满足以下规律:

- **加法交换律** $a+b=b+a$
- **加法结合律** $a+(b+c)=(a+b)+c=a+b+c$
- **加法幂等律** $a+a=a$
- **乘法结合律** $a(bc)=(ab)c=abc$
- **分配律** $a(b+c)=ab+ac$

  $(a+b)c=ac+bc$

  $(a+b)(c+d)=a(c+d)+b(c+d)$

在路径表达式中乘法不满足交换律。上述路径表达式中,a、b、c 和 d 均表示控制流图中的弧,但也可以代表路径。例如,若

$$X=abe+def+ghi$$

$$Y=uvw+xy$$

则

$$X+Y=(abe+def+ghi)+(uvw+xy)=abe+def+ghi+uvw+xy$$

$$XY=(abe+def+ghi)(uvw+xy)$$

5.3.2 程序中路径数的计算

程序中到底有多少条路径?有多少条线性独立路径数?对于简单的小程序,比如图 5-4(a)所示的程序,这个问题很容易回答,它有 4 条路径,但如果程序有多个判断和多个循环,其路径数并不容易直接从控制流图中看出,比如图 5-3 中的程序。下面讨论计算程序中路径数的一般方法。

1. 路径表达式计算

如果已经得到了程序的路径表达式,则可把其中的所有弧都用数值'1'代替,然后进行表达式的乘法和加法运算,即得该程序的路径数。

例如,图 5-4(a)所示的控制流图其路径表达式为 $e(a+b)(c+d)f$。将弧 e、b、c、d、f 均以 1 代入,表达式的值即为程序的路径数 N,N 的值为:

$$N=1\times(1+1)(1+1)\times1=4$$

再以图 5-4(c)为例。如果暂不考虑循环,那么从结点 1 到结点 5,只有两条路径,即

$$L=a(b+cd)e$$

代入数值 1,得到:

$$N=1\times(1+1\times1)\times1=2$$

再加上循环,这时 L 表示循环体。假定只考虑循环次数小于 3 的情况,路径表达式为:

$$L+LfL+LfLfL=L(1+fL+(fL)^2)$$

此式中 f 代入 1,$L=a(b+cd)e=2$ 则计算出路径数为:

$$2\times(1+1\times2+(1\times2)^2)=14$$

2. McCabe 的程序独立路径数

显而易见,程序中含有的路径数和程序的复杂性有着密切的关系,也就是说程序越复杂,它的路径数就越多。但程序复杂性如何度量呢?McCabe 给出了程序结构复杂性的计算公式。

程序控制流图是一个有向图,如果图中任何两个结点之间都至少存在一条路径,这样的图称

为强连通图。McCabe 提出,如果程序控制流图是一个强连通图,其复杂度 V(G)可按以下公式计算:

$$V(G) = e - n + 1$$

其中,e 为图 G 中的边数,n 为图 G 中的结点数,并且 McCabe 认为,强连通图的复杂度V(G)就是图中线性独立环路的数量。

通过从汇结点到源结点添加一条边,便可创建控制流图的强连接有向图。图 5-5 是一个经过了这种处理后的强连接有向图。其复杂度是:

$$V(G) = e - n + 1 = 11 - 7 + 1 = 5$$

图 5-5 中的强连接图的复杂度是 5,因此图 5-5 中有 5 个线性独立环路。如果现在删除从结点 G 到结点 A 所添加的边,则这 5 个环路就成为从结点 A 到结点 G 的线性独立路径。

以下给出用结点序列表示的 5 条线性独立路径:

p1 = A,B,C,G

p2 = A,B,C,B,C,G

p3 = A,B,E,F,G

p4 = A,D,E,F,G

p5 = A,D,F,G

独立路径是指从程序入口到出口的多次执行中,每次至少有一个语句(包括运算、赋值、输入、输出或判断)是新的,未被重复的。如果用前面提到的控制流图来描述,独立路径就是在从入口进入控制流图后,至少要经历一条从未走过的弧。

因此,路径p6 = A、B、C、B、E、F、G

p7 = A、B、C、B、C、B、C、G

不是独立路径。因为 p6 可以由路径 p1、p2 和 p3 组合而成,p7 可由路径 p1 和 p2 组合而成。

很明显,从测试角度来看,如果某一程序的每一条独立路径都测试过了,那么可以认为程序中的每个语句都已检验过了。但在实际测试中,要真正构造出程序的每条独立路径,并不是一件轻松的事。

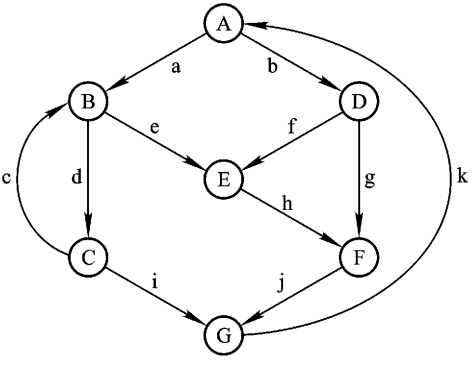


图 5-5 控制流图导出的强连通图

5.3.3 Z路径覆盖

结构测试覆盖中,人们最常用也是最为熟悉的测试覆盖准则是语句覆盖、分支覆盖、条件覆盖以及判定/条件覆盖等。至于路径覆盖,由于路径数目太多,使得人们不敢问津。

为了解决这一问题,必须舍掉一些次要因素,限制循环的次数。无论循环的形式和实际执行循环体的次数是多少,这里只考虑循环体执行一次和零次两种情况,也即只考虑执行进入循环体一次和跳过循环体这两种情况。这样可以极大地减少路径数量,使得覆盖这些有限的路径成为可能。这种简化循环意义下的路径覆盖称为Z路径覆盖。

图5-6(a)和图5-6(b)表示了两种最典型的循环控制结构。图5-6(a)先作判断,循环体B可能被执行,也可能不被执行,限定循环只执行一次或零次,这就和图5-6(c)所表示的条件结构一样。图5-6(b)先执行循环体B(也假定只执行一次),再经判断转出,其效果也与图5-6(c)中给出的条件选择结构只执行右分支的效果一样。

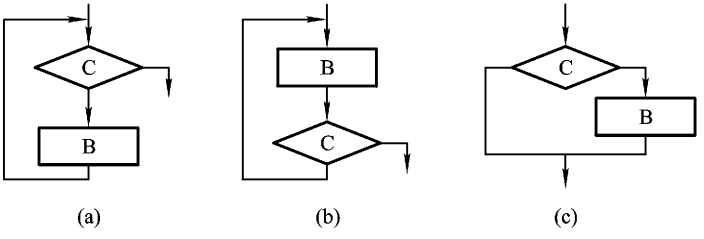


图5-6 循环结构和条件结构

5.3.4 独立路径测试

独立路径测试是在程序控制流图的基础上,通过分析控制结构的环路复杂性,导出可执行的独立路径集合,从而设计相应的测试用例。设计出的测试用例要保证被测程序的每条可执行的独立路径至少被执行一次。路径测试考虑以下几个方面:

- 程序控制流图。
- 程序环路复杂性。借助McCabe复杂性度量,可以从程序的环路复杂性导出程序路径集合中的独立路径条数。
- 设计测试用例。确保独立路径集中的每一条路径被执行。

由于测试用例要完成某条程序路径的执行,因此测试用例和测试用例所执行的程序路径之间有着非常明确的关系。

在路径测试中,最关键的问题仍然是如何设计测试用例,使之既能够避免测试的盲目性,又能有较高的测试效率。一般有3个途径可得到测试用例:

(1) 通过非路径分析得到测试用例

测试人员凭经验设计测试用例或由应用系统本身提供测试用例。在使用这些测试用例执行被测程序后,一些路径就被检测过了。

(2) 对未测试的路径生成相应的测试用例

枚举被测程序所有可能的独立路径,并与前面已测试过的路径相比,便可得知哪些路径还没

有被测试过,针对这些路径生成测试用例,进而完成对它们的测试。

(3) 生成指定路径的测试用例

根据指定的路径,生成相应的测试用例。

按以上方法实施测试,原则上是可以做到路径覆盖的,因为:

- 对程序中的循环作了如上限制以后,程序路径的数量是有限的。
- 程序的路径可经枚举全部得到。
- 完成若干个测试用例后,对所测路径、未测路径是知道的。
- 在指出要测试的路径以后,可以自动生成相应的测试用例。

5.4 数据流测试

数据流测试是指关注变量定义点和使用(或引用)点的一种结构测试方式,它和数据流图没有什么联系。实际上,很多数据流测试支持者和研究人员将这种测试方法看做是一种路径测试。从 20 世纪 60 年代初以来,人们一直通过分析变量的定义和使用,来查找如引用未定义变量等程序错误,也可用来查找对以前未曾使用变量的再次赋值等数据流异常的情况。找出这些错误很重要,因为这常常是常见程序错误的表现形式,如错拼名字、名字混淆或是丢失了语句等。下面首先来说明数据流分析的原理,然后介绍一种定义/使用数据流测试方法。早期的数据流分析常常集中于现在叫做定义/引用异常的缺陷,如:

- 变量被定义,但是从来没有被使用(引用)。
- 所使用的变量没有被定义。
- 变量在使用之前被再次定义。

这些异常可以通过程序的索引表发现。由于索引表信息是由编译器生成的,因此这些异常可以通过所谓静态分析发现,即在不执行被测程序的情况下发现源代码的一些数据流异常。

5.4.1 数据流分析

数据流分析在软件开发、测试和维护中起着十分重要的作用。它将程序中变量的出现分为变量的定义和引用。若语句 k 执行时改变了程序变量 v 的值,则称 k 定义了变量 v ;若语句 k 执行时引用了变量 v 的值,则称 k 引用了变量 v。

所谓数据流分析是指在不运行被测程序的情况下,对变量的定义、引用进行分析,以检测数据的赋值与引用之间是否出现了不合理的现象,如引用未赋值的变量,对以前未曾引用变量的再次赋值等数据流异常现象。

例如,语句

```
V = Y + Z ;
```

定义了变量 V,引用了变量 Y 和 Z,而语句

```
if (Y > Z) then. . . ;
```

则引用了变量 Y 和 Z。输入语句

```
Input  X
```

定义了变量 X。输出语句

output Y

则引用了变量 Y。执行某个语句也可能使变量失去定义，成为无意义的。例如，语句

while(X > 10)

中，循环控制变量 X 经循环正常出口离开循环时，就变成无意义的变量。

图 5 - 7 给出了一个小程序的控制流图，其中每个语句的定义/引用变量由表 5 - 5 给出。从表 5 - 5 可以看出，语句 1 定义了变量 X、Y 和 Z，表明它们的值是程序外赋给的，例如，该程序是以此 3 个变量为输入参数的过程或子程序。同样，出口语句 10 引用了 Z 变量，表明 Z 的值被送给外部环境。

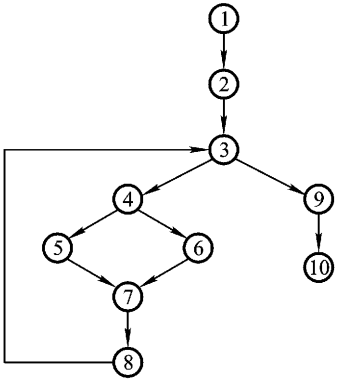


图 5 - 7 程序控制流图

表 5 - 5 变量的定义和引用表

结点	被定义的变量	被引用的变量
1	X,Y,Z	
2	X	W,X
3		X,Y
4		Y,Z
5	Y	V,Y
6	Z	V,Z
7	V	X
8	W	Y
9	Z	V
10		Z

通过变量的定义/引用分析，可以发现该程序中含有几个数据流异常：

① 语句 2 使用了变量 W，而在此之前并未对其进行定义(赋值)。

② 语句 5、6 使用了变量 V，但在第一次执行循环时也未对其定义过。

③ 语句 6 对变量 Z 的定义从未被使用过。

④ 语句 8 对 W 的定义也从未被使用过。

当然,程序中包含某些异常不一定会引起程序失效。但这种情况表明,也许程序中存在故障,应该仔细检查被测程序。

*5.4.2 定义/使用测试

假设 V 是程序 P 中变量的集合,程序 P 的控制流图用 G(P)表示。G(P)有一个单入口和一个单出口结点,并且不允许有从某个结点到其自身的边。为描述定义/使用测试,下面先定义几个基本的术语:

(1) 变量 v 的定义结点 n——记做 DEF(v,n)

结点 $n \in G(P)$ 是变量 $v \in V$ 的定义结点,当且仅当变量 v 的值由对应结点 n 的语句所定义。输入语句、赋值语句、循环控制语句和过程调用,都可以定义变量。如果执行了对应这些语句的结点,那么与被定义变量关联的存储单元的内容就会改变。

(2) 变量 v 的使用结点 n——记做 USE(v,n)

结点 $n \in G(P)$ 是变量 $v \in V$ 的使用结点,当且仅当变量 v 的值在对应结点 n 的语句中被引用。输出语句、赋值语句、条件语句、循环控制语句和过程调用,都是使用变量的结点。但是,执行对应这些语句的结点时,并不改变与被引用变量关联的存储单元的内容。

(3) 谓词使用——记做 P-use

USE(v,n)是一个谓词使用,当且仅当语句 n 是谓词语句;否则,USE(v,n)是计算使用,记做 C-use。

条件语句、循环控制语句中变量的使用一般是谓词使用,而赋值语句中变量的使用一般是计算使用。

(4) 定义/使用路径——记做 du-path

如果对某个变量 $v \in V$,存在一个定义、使用结点对,即 DEF(v,m)和 USE(v,n),使得变量 v 在结点 m 处被定义,在结点 n 处被使用,则从 m 到 n 的结点序列称为一条定义/使用路径,结点 m 称为该定义/使用路径的开始结点,而结点 n 则称为该定义/使用路径的结束结点。

(5) 定义明确路径——记做 dc-path

如果对某个变量 $v \in V$ 存在一个定义,使用结点对 DEF(v,m)和 USE(v,n),使得变量 v 在结点 m 处被定义,在结点 n 处被使用,并且从 m 到 n 的结点序列中没有其他结点对变量 v 进行过定义,则从 m 到 n 的结点序列称为一条定义明确的路径,结点 m 称为该定义明确路径的开始结点,而结点 n 则称为该定义明确路径的结束结点。

定义/使用路径和定义明确路径描述了变量从被定义点到被引用点的数据流向。不是定义明确的定义/使用路径,很可能是潜在问题的根源所在,所以应特别关注这些定义/使用路径。

1. 雇佣金问题的定义/使用路径测试

下面用雇佣金问题及其程序控制流图来说明这些定义。雇佣金问题的类 C 语言伪代码实现如下,其程序控制流图见图 5-8 所示。雇佣金问题根据所销售的枪机、枪托和枪管总数确定销售额,并计算雇佣金。其中的 while 循环当枪机值为 -1 时,结束接受数据。枪机、枪托和枪管的总销售量在 while 循环中通过累加数据值得到。打印出一些初步信息之后,利用程序开头部

分定义的商品价格计算销售额,然后,根据销售额计算雇佣金。

```
/* Program Commission */
1 main()
 {
2 int locks,stocks,barrels
3 float lockPrice,stockPrice,barrelPrice
4 int totalLocks,totalStocks,totalBarrels
5 float lockSales,stockSales,barrelSales
6 float sales,commission
7 lockPrice =45.0
8 stockPrice =30.0
9 barrelPrice =25.0
10 totalLocks =0
11 totalStocks =0
12 totalBarrels =0

13 scanf(locks)
14 while NOT(locks = -1)          \* loop condition uses -1to indicate end of data * \
15    scanf(stocks,barrels)
16    totalLocks = totalLocks + locks
17    totalStocks = totalStocks + stocks
18    totalBarrels = totalBarrels + barrels
19    scanf(locks)
20 endWhile
21 printf("Lockssold = ",totalLocks)
22 printf("Stockssold = ",totalStocks)
23 printf("Barrelssold = ",totalBarrels)
24 lockSales = lockPrice * totalLocks
25 stockSales = stockPrice * totalStocks
26 barrelSales = barrelPrice * totalBarrels
27 sales = lockSales + stockSales + barrelSales
28 printf("Total Sales = ",sales)
29 if(sales >1800.0)
30 then
31          commission =0.10 * 1000.0
32          commission = commission +0.15 * 800.0
33          commission = commission +0.20 * (sales -1800.0)
34 else if (sales >1000.0)
```

```
35        then
36              commission = 0.10 * 1000.0
37              commission = commission + 0.15 * (sales - 1000.0)
38        else commission = 0.10 * sales
39        endlf
40  endlf

41  printf("Commission is",commission)
42  end Commission
```

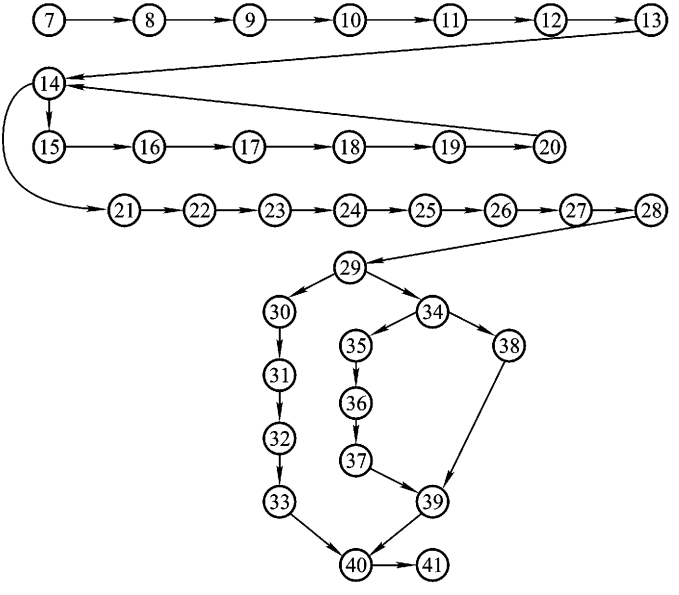


图 5-8 雇佣金问题的程序控制流图

表 5-6 给出了雇佣金问题中变量的定义结点和使用结点。使用这些信息,结合图 5-8 中的程序控制流图,可以识别各种定义/使用路径和定义明确路径。对于不可执行的语句,例如常量和变量说明语句,是否应该被认为是定义结点,现在还是学术界争论的一个问题。如果沿定义/使用路径跟踪程序的执行情况,则这些结点并不很重要。但是如果出现了问题,包含这些结点有助于发现问题,则可视情况做出选择。表 5-6 中没有考虑变量说明语句。

表 5-6 雇佣金问题中变量的定义/使用结点

变 量	定义结点	使用结点
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10,16	16,21,24

续表

变 量	定义结点	使用结点
totalStocks	11,17	17,22,25
totalBarrels	12,18	18,23,26
locks	13,19	14,16
stocks	15	17
barrels	15	18
lockSales	24	27
stockSales	25	27
barrelSales	26	27
sales	27	28,29,33,34,37,38
commission	31,32,33,36,37,38	32,33,37,41

表5-7列出了雇佣金问题中的部分定义/使用路径,其第三列表示定义/使用路径是否为定义明确路径。有些变量的定义/使用路径很简单,例如lockPrice、stockPrice和barrelPrice变量。有些变量的定义/使用路径要复杂一些,例如while循环(结点序列<14,15,16,17,18,19,20>)的循环控制变量locks和totalLocks、totalStocks和totalBarrels变量。由于篇幅的限制,这里只给出totalStocks变量的细节。totalStocks的初始值定义出现在结点11上,在结点17上第一次使用。因此,由结点序列<11,12,13,14,15,16,17>组成的路径(11,17)是定义明确。由结点序列<11,12,13,(14,15,16,17,18,19,20)*,21,22>组成的路径(11,22)就不是定义明确路径,因为变量totalStocks在结点11和17处被定义两次(可能在结点17处多次定义)。*号表示零次或多次重复。

表5-7 雇佣金问题中的部分定义/使用路径

变 量	路径(开始、结束)结点	是否定义明确路径
lockPrice	7,24	是
stockPrice	8,25	是
barrelPrice	9,26	是
totalStocks	11,17	是
totalStocks	11,22	否
totalStocks	11,25	否
totalStocks	17,22	否
totalStocks	17,25	否
locks	13,14	是
locks	19,14	是
locks	13,16	是
locks	19,16	是

续表

变　量	路径(开始、结束)结点	是否定义明确路径
sales	27,28	是
sales	27,29	是
sales	27,33	是
sales	27,34	是
sales	27,37	是
sales	27,38	是

下面较详细地讨论一些定义/使用路径。

(1) 变量 stocks 的定义/使用路径

首先研究一条简单的路径——变量 stocks 的定义/使用路径。这里有 DEF(stocks,15)和 USE(stocks,17),因此路径 <15,17> 是一个关于 stocks 的定义/使用路径。在路径 <15,17> 上,stocks 没有其他的定义结点,因此这条路径是定义明确路径。

(2) 变量 locks 的定义/使用路径

变量 locks 有两个定义结点和两个使用结点 DEF(locks,13)和 DEF(locks,19)以及 USE(locks,14)和 USE(locks,16),产生了 4 条定义/使用路径:

p1 = <13,14>

p2 = <13,14,15,16>

p3 = <19,20,14>

p4 = <19,20,14,15,16>

定义/使用路径 p1 和 p2 在结点 13 读入(定义)locks 的值,locks 在结点 14(while 语句)中有一个谓词使用 P - use,如果 while 语句为真(例如路径 p2),则在语句 16 处 locks 变量有一个计算使用 C - use。定义/使用路径 p3 和 p4 在结点 19(while 循环接近结尾处)读入 locks 的值。如果“扩展”路径 p1 和 p3 已包含结点 21,则有

p1′ = <13,14,21>

p3′ = <19,20,14,21>

则 p′1、p2、p′3 和 p4 构成 while 循环的非常完备的测试用例集合——旁路循环(不执行循环体)、开始循环、退出循环和重复循环。所有这些定义/使用路径都是定义明确路径。

(3) 变量 totalLocks 的定义/使用路径

totalLocks 的定义/使用路径会产生典型的计算使用测试用例。通过两个定义结点 DEF(totalLocks,10)和 DEF(totalLocks,16)及 3 个使用结点 USE(totalLocks,16)、USE(totalLocks,21)和 USE(totalLocks,24),可得到 6 条定义/使用路径。

p5 = <10,11,12,13,14,15,16>

是一条定义/使用路径,其中的 totalLocks 初始值有一个计算使用。这条路径是定义明确的。

p6 = <10,11,12,13,14,15,16,17,18,19,20,21>

忽略了 while 循环重复的可能性。子路径 <16,17,18,19,20,14,15> 可以多次经过就说明了这一点。路径 p6 是定义/使用路径但不是定义明确路径。因为结点 16 再次定义了变量 total-

Locks。

p7 = <10,11,12,13,14,15,16,17,18,19,20,21,22,23,24>

包含了路径 p6,也可以通过使用路径名称来表示,比如,路径 p7 可以表示为:

p7 = <p6,22,23,24>

定义/使用路径 p7 不是定义明确路径,它包含了定义结点 16。

p8 = <16,16>

实际上在结点 16 处的赋值语句右边引用的是结点 10 定义的值,所以,这种情况不允许作为定义/使用路径。

另外两条定义/使用路径都是 p7 的子路径:

p9 = <16,17,18,19,20,21>

p10 = <16,17,18,19,20,21,22,23,24>

这两条定义/使用路径都是定义明确的,并且都有前面讨论过的循环迭代问题。

(4) 变量 sales 的定义/使用路径

变量 sales 只使用了一个定义结点,因此关于 sales 的所有定义/使用路径都必须是定义明确的。之所以对这些定义/使用路径感兴趣,是因为它们可用来说明谓词使用和计算使用。在下面 sales 变量的 3 条定义/使用路径

p11 = <27,28>

p12 = <27,28,29>

p13 = <27,28,29,30,31,32,33>

中,路径 p13 是一个有 3 个使用结点的定义明确路径,它还包含路径 p10 和 p11。如果要测试 p13,还会知道 p13 是否覆盖了其他两条路径。

以路径 p12 开始的定义/使用路径有两种选择 选择路径 <27,28,29,30,31,32,33> 或选择路径 <27,28,29,34>。这里考虑路径 <27,28,29,34>,那么 sales 的剩余定义/使用路径是:

p14 = <27,28,29,34>

p15 = <27,28,29,34,35,36,37>

p16 = <27,28,29,38>

2. 定义/使用路径测试覆盖

假设已识别了所有程序变量的定义结点和使用结点及各个变量的定义/使用路径,Rapps-Weyuker 定义了一组数据流测试覆盖准则。在下面的数据流测试覆盖定义中,P 是被测程序,G(P) 是其控制流图,T 是 G(P) 的一个路径集合,V 是 P 中变量的集合并假设定义/使用路径都是可执行路径。

(1) 所有定义覆盖准则

集合 T 满足程序 P 所有定义覆盖准则,当且仅当所有的变量 $v \in V$,T 包含了从变量 v 的每个定义结点到 v 的一个使用结点的定义明确路径。

(2) 所有使用覆盖准则

集合 T 满足程序 P 的所有使用覆盖准则,当且仅当所有的变量 $v \in V$,T 包含了从 v 的每个定义结点到 v 的所有使用结点的定义明确路径。

(3) 所有谓词使用/部分计算使用覆盖准则

集合 T 满足程序 P 的所有谓词使用/部分计算使用覆盖准则，当且仅当所有的变量 $v \in V$，T 包含了从 v 的每个定义结点到 v 的所有谓词使用结点的定义明确路径，并且如果 v 的一个定义没有谓词使用结点，则定义明确路径至少包含一个 v 的计算使用。

（4）所有计算使用/部分谓词使用覆盖准则

集合 T 满足程序 P 的所有计算使用/部分谓词使用覆盖准则，当且仅当所有的变量 $v \in V$，T 包含了从 v 的每个定义结点到 v 的所有计算使用结点的定义明确路径，并且如果 v 的一个定义没有计算使用结点，则定义明确路径至少包含一个 v 的谓词使用。

（5）所有定义/使用路径覆盖准则

集合 T 满足程序 P 的所有定义/使用路径覆盖准则，当且仅当所有的变量 $v \in V$，T 包含了从 v 的每个定义结点到 v 的所有使用结点的定义明确路径，并且这些路径要么有一个环路，要么没有环路。

图 5－8 给出这些数据流测试覆盖准则之间的关系。由此可知，定义/使用测试提供了一种介于所有路径覆盖准则（一般认为是不能达到的）和所有结点（一般认为是最低的语句覆盖准则）之间的、更细化的结构测试可能性。可以说，定义/使用测试提供了一种检查故障可能发生点的严格的和系统化的方法。 图 5－9 数据流覆盖准则的层次结构

5.5 符号测试

测试用例生成是软件测试的核心和关键。但是,由于测试人员的实践经验不足和被测程序逻辑上的复杂性,无论是用黑盒测试还是白盒测试方法都不能保证选取到完全有代表性的测试数据。能不能避开这个关键问题呢,符号测试方法力图另辟捷径。

1. 符号测试

符号测试的基本思想是允许程序的输入不仅可以是具体的数值数据,而且还可以包括符号值,这里所说的符号值可以是初等符号值,也可以是符号表达式。初等符号可以是任何表示变量值的字符串,表达式可以是数、算术运算符和符号值的组合。这样,在执行程序过程中以符号计算代替了普通执行中的数值计算,所得到的结果自然是符号公式或是符号谓词。更明确地说,普通测试执行的是算术运算,符号测试执行的则是代数运算。因此符号测试可以认为是普通测试的一个自然扩充。

采用普通测试方法检测被测程序时,要从输入变量 X 的取值范围中选取一个具体的值进行数值运算。采用符号测试,只需使用符号值,例如以 x1 作为输入数据,代入程序进行代数运算,所得结果是一个含有 x1 的代数表达式。它的正确性对于判断程序的正确性就直观多了,因为这个代数表达式本身就表明了其运算过程,并且一次符号测试的结果代表了一类普通测试的运行结果。比如,上述对 x1 的测试也许等价于进行了 1 000 次普通测试(假定输入变量 X 的取值范围为 1 ~ 1 000),因此测试成本较低。

例如,下面一个小程序:

```
Proc(X,Y)
 {
   S = 2 * X + 3 * Y
    T = S - Y
    RETURN
}
```

如果将 a、b 作为程序变量 X、Y 的值,则有:

$$S = 2 * a + 3 * b$$

$$T = 2 * a + 3 * b - b$$

简化得到:$T = 2 * a + 2 * b$

2. 符号执行树

以符号值作为输入数据,在执行程序的过程中,如果遇到条件语句:

if (谓词) then... else...

则谓词可能是符号表达式,无法确定它究竟应该取真值,还是应该取假值,也就无法决定应该执行哪一个分支,这时两个可能的分支都必须保留,并分别做进一步的探索。如果此时能够通过其他信息,获得程序变量的取值情况,则可根据其做出选择。但通常的情况是不得不兼顾两条分支,分别继续执行。在符号执行的过程中,凡遇到条件判断,都会出现类似的情况。每遇到一个条件判断出现两个分支,分别继续下去。如果程序中仅有 if 语句构成的分支,那么符号执行的

过程可用一个二叉树来描述,称此二叉树为符号执行树。把路径上各分支点的谓词条件用逻辑乘符号联接起来,得到一个逻辑表达式,称为该路径的路径条件。

下面结合计算最大公因数的程序,对符号执行树和路径条件做进一步说明。图 5 – 10 给出了计算整数 X 和 Y 最大公因数的程序。图 5 – 11 则是该程序的符号执行树,树上有编号的结点对应于流程图的特定语句。除去分支结点外,普通结点都有一个后继结点。在分支点上,由于无法决定谓词的取值,只好两个分支均予保留,分别标明它所依赖的条件。例如,结点 3 的 $x \neq y$ 和 $x = y$。从程序入口处到条件语句之前符号执行是无条件的,可以在相应的控制流线上标以符号“T”。从树的根结点到每片叶子结点都构成了程序的一条路径,控制流上注明的是路径条件。

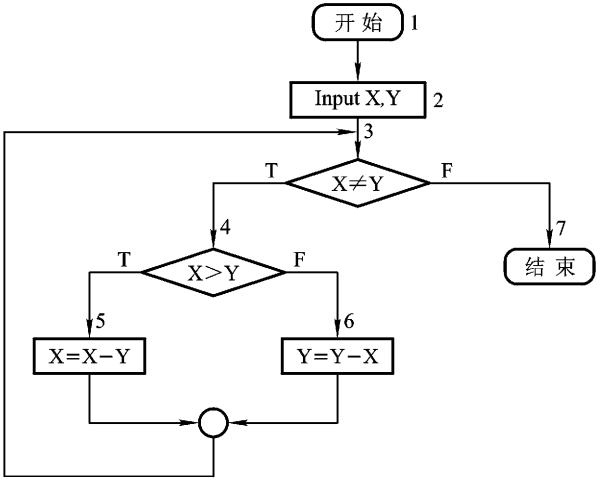


图 5 – 10 求最大公因数程序控制流图

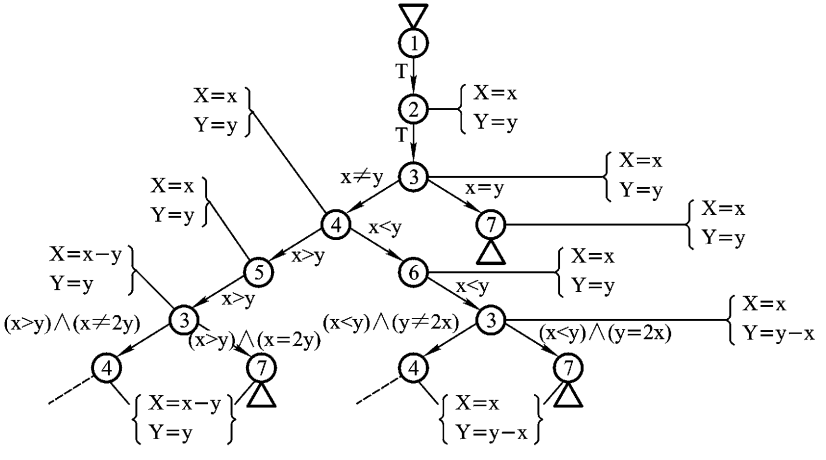


图 5 – 11 求最大公因数程序的符号执行树

从这个例子中可以看出,在程序的分支处,一个分支谓词的符号值实际上是对输入数据的限制要求。路径条件是该路径的谓词系统,它由多个等式或不等式组成。通过求解不等式,得到满足路径上各限制谓词的测试数据。若不等式无解,则相应的路径为不可行路径,这便是符号测试

的意义所在。

3. 符号测试评述

符号测试可以看做是测试与验证的一个折中方法。一方面,它沿用了传统的程序测试方法,通过运行被测程序来检验它的可靠性。另一方面,由于一次符号测试的结果代表了一大类普通测试的运行结果,实际上是证明了程序接受此类输入,所得输出是正确的,还是错误的。最为理想的情况是,程序中仅有有限的几条执行路径,如果对这有限的几条路径都完成了符号测试,我们就有较大的把握确认程序的正确性了。

从使用符号测试方法看,问题的关键在于是否能够开发出比传统编译器功能更强,并且能够处理符号运算的编译器或解释器。

在实际运用时,符号测试可按以下步骤进行:

① 利用符号执行解释器对被测程序进行符号执行。

② 若是遇到程序不能继续执行的情况,要求用户干预或是遍历执行树的各分支路径。

③ 化简得到的路径条件。

④ 用解线性不等式方法求解路径条件,以求得满足各个限制谓词的测试数据。

⑤ 若上述不等式无解,则相应的路径不可执行。

⑥ 对可行路径进行测试。

从以上的分析看出,符号测试方法的一个优点是,可以较容易地确定所给的一组测试用例是否覆盖了程序的各条路径。对于任何一组测试用例,可以首先确定它要经历的测试路径,然后,再给出输入变量的符号值,得到其路径条件。

假设程序有 n 条路径,其路径条件分别为 Pl、P2、...、Pn,其中

$$Pi == C1 \wedge C2 \wedge \ldots \wedge Cm。$$

Cj 为第 j 个分支上的谓词($1 \leqslant j \leqslant m$)。

若对于程序中的全部路径:

$$P = P1 \vee P2 \vee \ldots \vee Pn$$

测试用例 t1,t2,...,tn 恒为真,则表明程序中所有可能的控制路径都已检测过。否则,$\neg P$ 是没有走过的路径所对应的输入空间。

符号测试方法在使用中会遇到一些问题,比如在遇到循环、过程调用、动态数据结构、数组和指针处理时,符号执行实现困难。这些问题到目前为止尚未得到圆满的解决,因而也就严重地影响了它的发展前景。

5.6 域测试策略

相对于程序的执行路径,Howden 认为程序中出现的错误可以分为 3 类:域错误、计算错误和丢失路径错误。如果程序控制流有错误,导致某一特定输入执行了一条错误路径,这种错误称为路径错误,也叫做域错误。如果对于某一特定输入,程序执行的是正确路径,但由于赋值语句的错误致使输出结果不正确,这种错误则称为计算错误。另外一类错误是丢失路径错误,它是由于程序中某处少了一个判定谓词而引起的。对大多数域错误,Howden 认为符号测试和实际数据测

试是不可靠的,只有当测试人员幸运地从不正确的定义域部分选取一个测试数据时,实际数据测试才能发现这些错误。

针对程序域错误,White 提出了域测试策略并在以后发展成为一个有效的模块测试方法。域测试的‘域’指的是程序的输入空间。自然,每个被测程序都有一个输入空间,而输入空间又可以分为不同的子空间,每一子空间对应一个路径定义域。在考察被测程序的结构后,我们发现:子空间的划分是由程序中分支语句中的谓词决定的。

利用‘边界附近的处理对程序故障更加敏感’这一特点,域测试策略在分析输入域的基础上,对每一子域的每一谓词边界,通过选取适当的测试点,对谓词边界附近的处理进行检测。

把路径上各条件语句的谓词用逻辑乘联接起来,得到的表达式称为路径条件。路径条件规定了一个路径定义域,它由使程序沿该路径执行的所有输入组成,其边界(boundary)由路径上各个谓词决定。每个谓词对应一个边界(border),它是路径定义域边界的一部分。若无特殊声明,域测试策略中所指边界为谓词对应的边界,简称谓词边界。

为测试一个谓词边界,域测试策略定义了两类测试点:一类称为 ON 测试点,位于被测谓词边界上;一类称为 OFF 测试点,离被测边界有一小段距离 ε,并在被测域之外。图 5-12 给出了 ON-OFF 测试点的选取原则。其中 AB 是一条谓词边界(由被测程序决定),PQ 是程序正确时实际谓词边界的位置。A、B 是两个 ON 测试点,C 点是一个 OFF 测试点,离 AB 边界的距离是 ε 如果将 ON 测试点与 OFF 测试点交替选择,即让 OFF 测试点在两 ON 测试点所决定直线上的投影在两 ON 测试点之间,则可较好地测出由于边界偏移而导致的域错误。这就是域测试的基本思想。

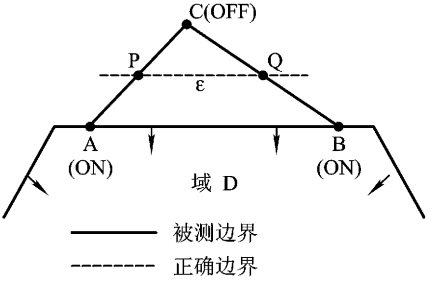


图 5-12 谓词边界测试点的选择

边界偏移是指错误边界离正确边界相差不大的情况。可以将边界偏移分为 3 种情况,如图 5-13 所示。图 5-13(a)中边界偏移使得域 D_1 减小,测试点 A、B 将给出正确结果,但 C 测试点将给出错误的结果。图 5-13(b)中边界偏移使得 D_1 增大,测试点 C 将给出正确结果,但测试点 A、B 将给出错误结果。在图 5-13(c)中,测试点 A、C 将给出正确结果,而测试点 B 将给出错误结果。这表明 ON-OFF-ON 这种交错选取法对边界偏移错误是相当有效的。可见,ON-OFF 测试点的选取,是域测试实现的关键。

判定测试点输出结果正确与否的依据是该测试点是否属于其应属于的域中。由图 5-12 可以看出,如果 A、B、C 3 个测试点输出结果不正确,显然程序有错。如果 3 个测试点均产生正确

的输出结果,则被测边界可能正确,也可能不正确。当正确的边界 PQ 介于 C 点与线段 AB 之间时,被测边界的偏移有可能检测不出来。因为这时,A、B 测试点位于一个域中,而 C 测试点位于其相邻的域中,它们都属于应属的域中,而 PQ 正好位于它们之间。如果将 ε 取得尽可能小,就可认定被测边界的正确性。

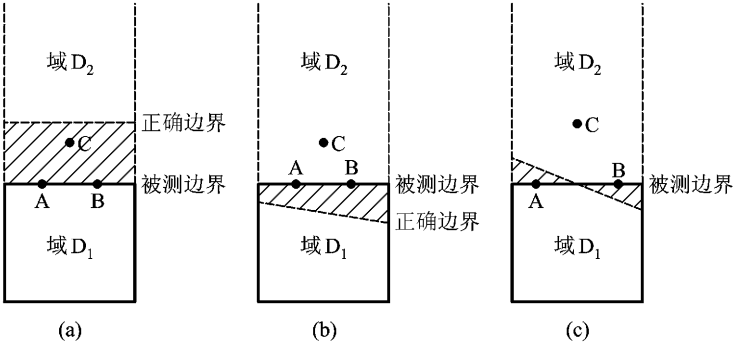


图 5 – 13 边界偏移的 3 种情况

同样,程序谓词中运算符的错误也能通过这种域测试策略检测出来,也就是说,常见的错误,比如该用‘ ≥ ’时,错用成‘ ≤ ”,该用‘ < ’处,错用成‘ ≤ ’等错误都能较容易地检测出。

以上是针对二维输入域的情况讨论的。当一个程序谓词与多个输入变量有关时,涉及多维超平面。域测试的基本思想仍然适用,但对于 n 维情况,需要找出 n 个线性无关的 ON 测试点和一个 OFF 测试点来测试一个谓词边界的偏移。通常把超平面的极值点选作 ON 测试点,以保证其线性无关。对于一个由 P 个谓词边界组成的子域,那么最多需要(N + 1)P 个测试点。尽管相邻边界的交点可以作为两个边界共同的 ON 测试点,但测试点的数目仍然较大,测试费用较高。

应当注意,以上讨论的边界都是由不等式谓词产生的,即 exp > 0、exp < 0、exp ≥ 0 以及 exp ≤ 0 之类的谓词。如果谓词形如 exp = 0 或 exp ≠ 0,则需要两个 OFF 测试点,两个 ON 测试点。两个 OFF 测试点分别在两个 ON 测试点连线的两侧,才能确保检测出谓词边界的偏移。

随后,Jeng 提出了一种简化的域测试策略,对于被测边界,它只要求选取一个 ON 测试点,一个 OFF 测试点。要求:

ON 测试点满足被测谓词边界的路径条件。OFF 测试点位于被测谓词边界之外,并且离 ON 测试点尽可能近。

测试一个不等式(关系运算符为 <、>、≤、≥)谓词边界时,需选取一个 ON 测试点和一个 OFF 测试点。测试相等或不等(关系运算符为 = 或 ≠)谓词边界时,需选取一个 ON 测试点和两个 OFF 测试点,两个 OFF 测试点分别位于被测边界的两侧,且离 ON 测试点尽可能近。

尽管域测试策略可有效地检测出由于谓词边界偏移而导致的域错误,但由于域测试策略的限制较多,另外当程序存在很多路径时,所需的测试点也很多,较难具体运用到实践中。

5.7 程序变异

程序变异是一种不同于黑盒测试和白盒测试的测试方法,它属于故障驱动测试。所谓故障驱动测试方法,是指该方法适用于某类特定的程序故障。显然,要想找出程序中所有的软件故障是不可能的。一种比较现实的办法是尽可能地缩小故障的搜索范围,以便于检测某类故障是否存在,这样可以集中精力寻找对软件危害较大的可能故障,提高测试效率,降低测试成本。

程序强变异和弱变异是两种主要的错误驱动测试方法。

5.7.1 程序强变异

程序强变异简称为程序变异。程序变异假定被测程序由经验丰富的程序人员编写,这样的程序即使不正确,残留在程序中的错误也不是那些重大的错误,而是一些难以发现的小错误。和正确软件相比,表现为一个符号或几个符号的错误。比如,遗漏了某个操作,分支谓词规定的边界有偏移等。即使是一些稍微复杂的错误,也可以看做是这些简单错误的组合。程序变异的目标就是查出这些简单的错误及其组合。

那么,什么是程序变异?假设对程序 P 进行了微小改动而得到程序 MP,则 MP 也是一个程序,称为 P 的一个变异体。

显而易见,如果程序 P 是正确的,MP 也是一个几乎正确的程序。如果 P 不正确,则 P 的某一个变异体 MP 可能是正确的。

假设为程序 P 设计了一组测试数据,记为 T,如果用这一组测试数据,检测到程序 P 中有故障,则可对程序 P 进行修改。如果 P 在 T 上运行时,没有发现故障,那么,不能肯定程序是正确的,但借助于程序变异思想,可以确定程序的正确程度。

如果 P 在 T 上是正确的,可以针对一些变异运算生成若干个只在个别部位与 P 有所不同的变异体,记为集合 M,即:

$$M = \{MP \mid MP \text{ 是 } P \text{ 的变异体}\}$$

如果 M 中的所有变异体在 T 上执行的结果都不正确(称为被‘杀’掉),那么 P 很可能是正确的,可以认为程序 P 的正确程度较高。如果 M 中某些变异体在 T 上运行正确,则可能:

① 现有的测试数据不足以找出 P 与其变异体之间的差别。

② P 可能含有故障,而其某些变异体却是正确的。

情况①说明,当 P 与某些变异体在 T 上运行结果都正确时,可能是因为测试数据太少或测试数据不够典型造成的。这时可以增加测试数据,直到‘杀掉’所有的变异体,也就是说,让所有的变异体都出错,这时可以认为 P 是正确的。情况②则提醒我们,当许多典型的测试数据仍然不能使某一变异体出错时,该变异体可能是程序的正确形式。

程序变异意味着对测试数据集中的每一元素都要对程序 P 及其变异体进行测试,所以测试数据集 T 和变异体集合 MP 都需精心挑选,这是强变异方法成功的关键。

使用程序变异方法,最重要的是如何建立变异体。变异体可以看做是变异运算作用在被测程序上的结果。变异运算接受被测程序为输入,而产生一系列不同的变异体。例如,可以将数据元素用其他的数据元素替换,可以将常量增加一点,或是减少一点,也可以将数组变量名替换为

另一个维数相同的数组变量名,还可以对操作符作某些变换,可以替换或删除语句,甚至改变条件语句的转向点等。

Budd 等人曾列出了一些常用的变异运算:

- 常量之间替换。
- 将常量替换为标量。
- 将常量替换为数组分量。
- 将标量替换为常量。
- 将标量替换为数组分量。
- 将数组分量替换为常量。
- 将数组分量替换为标量。
- 数组分量之间替换。
- 算术运算符替换。
- 关系运算符替换。
- 逻辑运算符替换。
- 插入绝对值符号。
- 插入单目运算符。
- 语句分解。
- 语句删除。
- 循环终止条件变换。

总之,对程序进行变换的方式有多种多样。究竟对程序进行什么样的变换,很多情况下,与测试人员的实践经验有关。实际上,通过变异分析构造测试数据的过程是一个循环过程。当对程序及其变异体进行测试后,如果有些变异体没有被‘杀掉’,这时可以增加测试数据,直到所有的变异体被‘杀掉’。

Budd 等人曾在 13 个被测程序(共有 30 个故障)上,分析了程序变异系统在辅助测试人员进行测试方面的效果。分析结果如表 5-8 所示。从中可以看出程序变异是一种较为有效的测试方法。

表 5-8 变异测试查错效果

故障类型	总 数	查 出 数
丢失路径错误	7	6
不正确谓词	5	3
不正确计算语句	15	14
丢失计算语句	3	2

程序变异方法的一个主要弱点是,要运行所有的变异体,从而成倍地增加了测试的成本。但是,程序变异由于其针对性强、系统性强,正逐渐成为软件测试中一种广泛使用的测试方法。特别是在变异测试系统的支持下,用户可以更有效地测试自己的程序。

*5.7.2 程序弱变异

程序弱变异也是一种变异方法,其目标仍然是要查出某一类故障。其主要思想是,假设 P 是一个程序,C 是 P 的一个简单组成部分,若有一变异变换作用于 C 而生成了 C′,如果 P′是含有 C′的 P 的变异体,则弱变异是要设计测试数据,使得当 P 在该测试数据下运行时,C 被执行,且至少在一次执行中,C 产生的值与 C′不同。

由此可以看出,弱变异是一种只针对被测程序进行测试的变异方法,它强调变动程序的组成部分,根据弱变异准则,只要能够设计出使得 C 与 C′产生不同值的测试数据,就可将被测程序在此测试数据上运行,并不需要实际产生其变异体。

实现弱变异的关键是,确定程序 P 的组成部分以及与其相关的变换。组成部分可以是程序中的计算结构、变量定义与引用、算术表达式、关系表达式以及布尔表达式等。其中一个组成部分可以是另一个组成部分的一部分,最基本的程序组成部分有:变量定义、变量引用、算术表达式、关系表达式和布尔表达式,其中布尔表达式可以包含关系表达式,关系表达式又可以包含算术表达式,布尔表达式、关系表达式和算术表达式可以包含变量定义和变量引用。下面分别进行简单讨论。

(1) 变量定义

变量定义指的是给变量赋以新值,如在语句 A = B 中,变量 A 被赋予变量 B 的值,该语句被称为变量定义组成部分。

变量定义的弱变异使被定义的变量变为另一变量。如在语句 A = B 中,使变量 A 变为另一程序变量 D,则语句变为 D = B。假设 V 是程序 P 中变量定义组成部分 C 的一个被定义变量,C′是 C 的一个变换,要使 C 和 C′产生不同的值,必须设计测试数据,使程序执行到 C 前 V 的值与执行 C 后 V 的值不一致,否则,即便 C 的定义是错误的,因 V 的值没有变化,因而也不容易查出这一错误。

(2) 变量引用

变量引用是指变量值的使用,如对于语句:

A = B + A

变量 B 被引用,变量 A 则先被引用,继而又被定义。再如语句:

IF(A > B)

中,A、B 的值均被引用,这些语句是变量引用的组成部分。

变量引用的弱变异使得被引用的变量变为另一变量。例如在语句 A = B + A 中,使 B 变为另一程序变量 D,则语句变为 A = D + A。假设 V 是程序 P 中变量引用部分 C 的一个被引用变量,C′是 C 的一个变换,要保证 C 和 C′在测试数据上执行时产生的值不同,要求程序执行到 C 时,所有程序变量的值都与 V 的值不同。这是因为,若组成部分 C 为 A = V,变换 C′为 A = W,如果执行到 C 时,W 的值与 V 相等,那么这组测试数据就查不出此变量的引用错误。

从以上的分析可以看出,要保证在一个变量引用中,被引用的变量是正确变量,必须设计一组测试数据,使得运行到此变量引用时,被引用变量与任何其他程序变量的值都不相等。一般而言,一组测试数据只能保证某几个程序变量值与被引用变量值在运行到变量引用部分时不相等。要让所有的程序变量值都与被引用变量值在运行到变量引用部分时不等,需要很多组测试数据。

实际上,由于程序中有很多变量引用组成部分,要对所有组成部分进行弱变异测试,是非常耗费时间的。

一种比较可行的办法是,将变量引用组成部分局限在错误数组元素引用和错误输入变量引用两种情况上。实际工作中,由于下标错误,常常会发生错误的数组元素引用;当函数之间有参数传递时,函数中的形式参数(它是被引用的),也常常会出现错误。对这两种情况进行弱变异的准则是:选取测试数据,使程序运行到该组成部分时,数组的各元素均不相等,或者是程序的形参互不相等。

(3) 算术表达式

算术表达式的弱变异考虑3种常见错误:表达式与正确表达式相差一个常数;表达式是正确表达式的常数倍;表达式的系数有错误。实际上,后一种情况包括了前两种情况。由于前两种情况要求较少的测试数据,因而将其单独分类。

(4) 关系表达式

关系表达式的弱变异考虑两类简单的错误:关系运算符错和差一个常数的错误。

(5) 布尔表达式

布尔表达式是将算术表达式用逻辑运算符(NOT、AND 和 OR)连接起来的复杂关系。

弱变异测试相对于强变异测试,可以看做是一种测试数据选择的准则,其优点是,用一组测试数据很可能就会检验出多个程序组成部分的错误,从而减少测试中程序执行的次数。强变异方法程序执行的次数却是随着变异体的增加而增加。另外,弱变异的准则对测试数据的选择有一定的指导作用,用户可以有针对性地选择测试数据。强变异方法对测试数据选取的指导作用不大。然而,弱变异测试也有它的不足之处,这表现在由于它重视的对象是程序的组成部分,所以即使当某一组成部分 C1 有错,而另有一组成部分 C2 也有错时,程序在能检查出 C1 错的测试数据下运行时,可能会由于 C2 的作用,使程序执行结果正确。强变异强调的是被测程序,它要求'杀掉'变异体,所以不会出现这种情况。

5.8 程序插装

程序插装是一种借助于向被测程序中插入操作来实现测试目的的方法,在软件测试中有着广泛的应用。

在调试程序时,常常需要在程序中插入一些打印语句,希望执行程序时,顺带打印出一些有用的信息,通过这些信息了解程序执行过程中的一些动态特性。比如,程序的实际执行路径,或是特定变量在特定时刻的取值。从这一思想发展出的程序插装技术能够按用户的要求,获取程序的各种信息,成为测试工作的有效手段。

比如,如果想了解一个程序在某次执行中所有可执行语句被覆盖(或被经历)的情况,或是每个语句的实际执行次数,最好的办法是利用插装技术。这里以计算整数 X 和整数 Y 的最大公约数程序为例,说明程序插装技术。图 5-14 给出了插装后的最大公约数程序的控制流图。图中的虚线框就是为了记录语句执行次数而插入的,其形式为:

$$C(i) = C(i) + 1 \qquad i = 1,2,\cdots,6$$

程序从入口开始执行到出口结束,经过的计数语句记录下该程序点的执行次数。如果在程序的

入口处插入了对计数器 C(i) 初始化的语句,在出口处插入了打印这些计数器的语句,就构成了完整的插装程序,它便能记录并输出在各程序点上语句的实际执行次数。

通过插入语句获取程序执行中的动态信息,这一做法如同在刚研制成的机器的特定部位安装记录仪表一样。安装好以后开动机器试运行,除了可以从检验机器加工的成品得知机器的运行特性外,还可以通过记录仪表了解其动态特性。相当于在执行程序以后,一方面可检验测试的结果数据,另一方面可以借助插入语句记录的信息了解程序的执行特性。

在程序的特定部位插入记录动态特性的语句,最终是为了把程序执行过程中发生的一些重要历史事件记录下来。例如,记录在程序执行过程中某些变量值的变化情况,变化的范围等。又如在本章 5.2 节中所讨论的程序逻辑覆盖情况,也只有通过程序插装才能获得覆盖信息。实践表明,程序插装方法是一种应用很广的技术,特别是在进行程序测试和调试时非常有效。

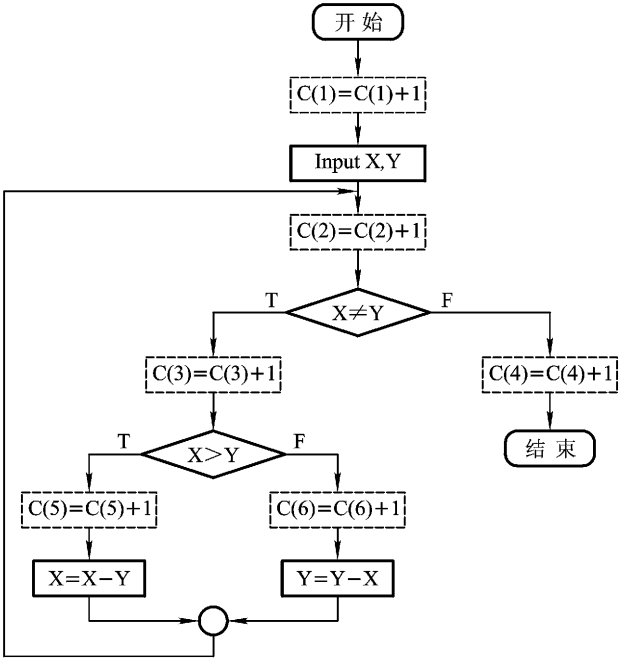


图 5-14 插装后的求最大公因数程序流程图

设计程序插装时需要考虑以下几个问题:

- 探测哪些信息。
- 在程序的什么部位设置探测点。
- 需要设置多少个探测点。

其中,前两个问题需要结合具体课题解决。至于第三个问题,需要考虑如何设置最少探测点的方案。例如,图 5-14 中程序入口处,若要记录语句 Input(X,Y) 的执行次数,只需插入 C(1) = C(1) + 1 这样一个计数语句就够了,没有必要在每个语句之后都插入一个计数语句。在一般情况下,可以认为,在没有分支的程序段中只需要插入一个计数语句。但程序中由于出现多种控制结构,使得整个结构十分复杂。为了在程序中设置最少的计数语句,需要针对程序的控制结构进行具体的分析。

软件测试中使用程序插装技术主要用于以下3个方面：

(1) 覆盖分析

程序插装可以用来确定和估计有关程序结构元素被覆盖的程度，从而确定测试执行的充分性，设计更好的测试用例，提高测试覆盖率。

(2) 监控和断言

在程序的特定部位插入某些用以判断变量特性的断言语句，以便证实程序运行时的某些特性，从而帮助排除故障。

(3) 查找数据流异常

数据流插装可以记录每个变量的最大值和最小值，从而发现超出预计范围的情况，还可以发现引用未经初始化的变量，以及已定义过但未曾使用的变量等数据流异常。虽然这些异常情况也可以由静态分析器发现，但使用程序插装的方法毕竟比较经济，因为所需的信息是在程序的测试执行中附带得到的，可以说是测试的副产品。

小结

白盒测试方法基于被测程序的源代码开发测试用例。常见的白盒测试方法有逻辑覆盖、数据流测试、域测试、符号测试、路径分析、程序变异以及程序插装等。

逻辑覆盖以程序内部的逻辑结构为基础设计测试用例，要求对被测程序的结构做到一定程度的覆盖，如语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖以及路径覆盖。路径覆盖是最强的逻辑覆盖准则，实际上只能有选择地测试程序中某些有代表性的路径。

独立路径选择和Z路经覆盖是两种常用的路径覆盖方法。

数据流测试是指关注变量定义点和使用(或引用)点的一种结构测试方式，用来检测数据的赋值与引用之间是否出现了不合理的现象，如引用未赋值的变量，对以前未曾引用变量的再次赋值等数据流异常现象。

符号测试允许程序输入符号值，以符号计算代替普通执行中的数值计算，但在遇到循环、过程调用、动态数据结构、数组和指针处理时，符号执行实现困难。

域测试策略在分析输入域的基础上，对每一子域的每一谓词边界，通过选取适当的ON，OFF测试点，对谓词边界附近的处理进行检测。

程序变异是一种不同于黑盒测试和白盒测试的测试方法，它属于故障驱动测试，适用于某类特定的程序故障。

程序插装是一种借助于往被测程序中插入记录语句来实现测试目的的方法，在软件测试中有着非常广泛的应用。

第5章习题

1. 白盒测试的最大问题是什么？
2. 本章描述的几种代码覆盖中哪一种最好，为什么？
3. 常用的白盒测试方法有哪些，它们的共同点是什么？

4. 试为三角形程序开发条件覆盖测试用例和判定/条件覆盖测试用例。注意:语句片段 11 ~ 15 之间表达式(a = b) AND(b = c)的依赖关系。

5. 仔细研究语句片段 11 ~ 15,对于 a = c 测试用例(例如,a = 4,b = 5,c = 4)会出现什么情况,第 11 行语句中的条件利用等式的传递性去掉了条件 a = c,这会产生问题吗?

6. 试从雇佣金问题的代码行 27 ~ 29、33、34、37、38 中找出关于销售额的定义明确路径。

第 6 章　集成测试与系统测试

单元测试工作完成后,对软件的测试才真正开始,特别是对于大型软件产品更是如此。根据软件故障的定义‘ 如果软件开发没有按用户的合理要求去做,则必定存在软件故障 ”,即使做到绝对完美的单元测试,也无法确保能够查出所有的软件故障。

实践表明,一些模块能够单独正常工作,并不能保证连接起来也能正常工作。程序在某些局部反映不出来的问题,在全局上很可能暴露出来,影响到功能的发挥。集成测试就是将多个模块组合在一起进行测试的过程。

通过单元测试和集成测试,仅能保证软件开发的功能得以实现。但不能确认在实际运行时,它能否满足用户的需求,是否存在实际使用条件下可能被诱发的故障隐患。为此,对完成开发的软件必须经过规范的系统测试。

本章重点:

- 自顶向下和自底向上的增式集成测试方法
- 常用的系统测试类型
- 配置测试
- 可用性测试
- 兼容性测试
- 文档资料测试
- 网站测试

6.1　集成测试

1999 年 9 月,火星气象人造卫星在经过 41 周 4.16 亿英里飞行后,在即将要进入火星轨道时失败了,为此,美国投资 5 万美元调查事故原因,发现太空科学家洛克希德 · 马丁采用的是英制(磅)加速度数据,而喷气推进实验室则采用的是公制(牛顿)加速度数据进行计算。如果他们进行了集成测试,事故本来是可以避免的。

在每个模块完成单元测试之后,需要着重考虑的一个问题是 :通过什么方式将模块组合起来进行集成测试,这将影响到模块测试用例的设计、所用测试工具的类型、模块编码的次序、测试的次序以及设计测试用例的费用和纠错的费用等。可见这是一个相当重要的问题。

6.1.1　增式集成测试与非增式集成测试

如何合理地组织集成测试,是首先独立地测试程序的每个模块,然后把它们组合成一个整体进行测试好,还是先把下一个待测试的模块组合到已经测试过的那些模块上去,再进行测试好呢?前一种方法称为非增式集成测试法(有时称为大爆炸测试方法),后一种方法叫做增式集成测试法。

图 6－1 给出了一个程序例子:图中的 7 个矩形分别表示程序的 7 个模块(子程序或者过程),模块之间的联线表示程序的控制层次,即模块 M1 调用模块 M2、M3 和 M4,模块 M2 调用模块 M5 和 M6 等。非增式集成测试方法先对 7 个模块中的每一个进行测试,可以同时测试或是逐个地测试各个模块,这主要由测试环境(如所用的计算机是交互式还是批处理式)和参加测试的人数等情况来决定。测试完毕后再把这些模块组合起来形成一个完整的程序进行测试。

测试每一个模块都需要一个专门的驱动模块和一个或多个桩模块。例如,为了测试模块 M2,首先要设计测试用例,然后由驱动模块把测试用例传送给模块 M2,作为其输入变量。驱动模块用以模拟被测模块的上级模块,用来驱动或传送测试用例给被测模块,还需向测试者显示执行模块 M2 后所产生的结果。此外,还需要模拟被测模块工作过程中所调用的模块,例如测试模块 M2 还需要模拟模块 M5 和 M6 的调用,这就是桩模块要做的工作。桩模块由被测模块调用,以便检验被测模块与其下级模块之间的接口。在 7 个模块全部测试完成后,再将它们联接起来,构成一个完整的程序进行测试。

图 6－2 给出了一个采用非增式集成测试方法的例子,图中标有 d 的模块表示驱动模块,标有 S 的模块表示桩模块。被测程序由图 6－1 所示的 7 个模块构成。在进行单元测试时,根据他们在结构图中的地位,对模块 M2 和 M4 配制了驱动模块和桩模块,对模块 M3、M5、M6 和 M7 只配制了驱动模块,对主模块 M1 配制了 3 个桩模块,以模拟被它调用的 3 个模块 M2、M3 和 M4。分别进行单元测试以后,再按图 6－1 的结构图形式联接起来,进行集成测试。

增式集成测试是另一种集成测试方法,它不是孤立地测试每一个模块,而是一开始就把待测试的模块与已测试过的模块联接起来。增量集成测试方法的种类很多,比如从程序底部开始(自底向上)测试。先由 4 个人并行或顺序地测试模块 M3、M5、M6 和 M7,这时,需为每个模块准备一个驱动模块,但不需要准备桩模块。然后测试模块 M2 和 M4,它们不是孤立地测试,而是把 M2 联在模块 M5 和 M6 上,模块 M4 联在模块 M7 上,也就是如果要测试模块 M2,先要设计一个驱动模块,再对模块对 M2～M5、M2～M6 进行测试。增式集成测试的过程,就是不断地把待测试的模块联接到已测试过的模块集(或其子集)上,对待测试模块进行测试,直到最后一个模块(这里是 M1)测试完毕。这个过程也可以自顶向下地进行。

从非增式集成测试和增式集成测试的集成过程,可以看出:

① 非增式集成测试方法工作量大。对图 6－2 而言,这个程序需要 6 个驱动模块和 6 个桩模块(假设最顶上的模块不需要驱动模块)。而自底向上的增式测试需要 6 个驱动模块,但不需要桩模块。因为非增式集成测试中所需要的驱动模块或者桩模块在增式集成测试中被已测模块所代替,增式集成测试使用了较少的辅助模块,因而增式集成测试的工作量减小了。

② 增式集成测试中,能够较早地检查出模块之间接口的错误。相反,若用非增式集成测试,在测试过程完成之前,各模块之间是没有联系的。

③ 增式集成测试,查找故障比较容易。非增式集成测试时先分散地进行测试,再集中起来一次完成集成测试。如果模块接口之间有故障,那么非增式集成测试只有当模块连接成完整的程序后,才能检查出这个故障。这时,由于错误可能在程序的任何地方,而且改正一个故障的同时又可能引入新的故障,新旧故障混杂,因而很难确定故障的原因和位置。相反,增式集成测试采用逐步集成和逐步测试的方法,发现的故障往往可能与刚联上的那个待测模块有关,检查就变

得容易了,故障易于定位和纠正。

④ 增式集成测试可以更彻底地对程序进行测试。在增式集成测试逐步集成的过程中,一些模块得到了较多的检验。比如,要测试模块 M2,就要执行模块 M5 和 M6,尽管前面已对模块 M5 和 M6 做了充分的测试,但在测试 M2 时,还要再一次执行模块 M5 和 M6,这时或许会产生新的情况,这些情况可能是原来测试模块 M5 和 M6 时所没有的。换言之,增式集成测试能够用已测试过的模块来代替一些驱动模块或者桩模块,这样,完成最后一个模块测试时,实际上各模块得到了更多的考验。

⑤ 非增式集成测试方法需要的机器时间较少。例如,如果用自底而上的增式集成测试方法来测试图 6 – 1 的模块 M1,那么在模块 M1 的测试过程中,模块 M2、M3、M4、M5、M6 和 M7 也许又被执行了一次,而用非增式集成测试方法来测试模块 M1,仅执行模块 M2、M3 和 M4 的桩模块。因此增式集成测试比非增式集成测试需要更多的机器指令。但是,非增式测试需要更多的驱动模块或桩模块,需要更多的时间来设计驱动模块或桩模块。

⑥ 采用非增式集成测试,所有模块可以同时进行并行测试,这对一个大型项目的测试非常重要。

总之,非增式集成测试和增式集成测试各有优缺点。上述几点中,前面 4 项是增式集成测试的优点,而后面两项是非增式集成测试的长处。但是,当前计算机工业硬件价格不断降低,而软件开发成本正在逐渐提高,尽早地发现软件故障,修复故障的成本就能够降低。因而增式集成测试的优点显得更加重要。所以,增式集成测试是一种更合适的测试方法。

6.1.2 自顶向下集成测试与自底向上集成测试

增式集成是构造程序结构的一种方式,按照不同的模块集成方式,又分为自顶向下增式测试和自底向上增式集成测试两种。自顶向下集成从主控模块开始,按照软件的控制层次结构,逐步把各个模块集成在一起。自底向上集成则从最下层的模块开始,按照程序的层次结构,逐渐形成完整的整体。

1. 自顶向下增式集成测试

自顶向下增式集成测试表示逐步集成和逐步测试是按程序结构图自上而下进行的,即从顶层主模块开始测试,这以后如何选择下一个要测试的模块并没有一个统一的方法。唯一的原则是:下一次要测试的模块,至少有一个调用它的模块已经测试过。

自顶向下增式集成测试的具体步骤是:

① 对主控模块进行单元测试,然后以主控模块作为测试驱动模块,把对主控模块进行单元测试时引入的所有桩模块逐渐用实际模块替代。

② 依据所选的集成策略,每次只替代一个桩模块。

③ 每集成一个模块立即测试一遍。

④ 只有每组测试完成后,才着手替换下一个桩模块。

⑤ 为避免引入新故障,需不断地进行回归测试(即全部或部分地重复已做过的测试)。从第二步开始,循环执行上述步骤,直至整个程序结构构造完毕。

图 6 – 3 给出了一个具有 12 个模块的程序流图,其中 M1 ~ M12 分别表示程序中的 12 个模块。采用自顶向下增式测试方法,第一步先测试模块 M1。要测试模块 M1,必须先编写出表示模块 M2、

M3 和 M4 的各个桩模块 S1、S2 和 S3，以模拟被 M1 调用的模块。参看图 6－4(a)。设计桩模块的工作并不是一件简单的工作。测试完 M1 后，将由下一个要测试的模块来代替其中一个桩模块，并加入这个模块所需要的桩模块。图 6－4(b)描述了程序自顶向下增式集成的下一步形式。

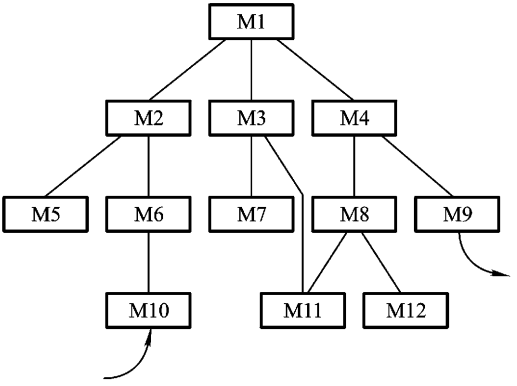
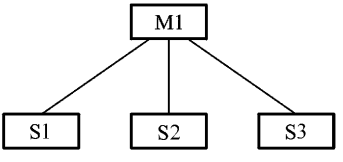
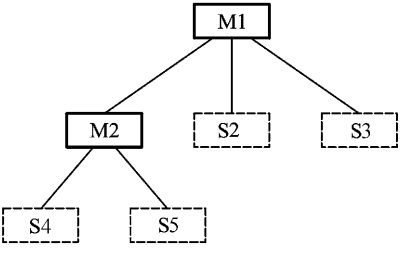


图 6－3　12 模块的程序流图



(a)



(b)

图 6－4　自顶向下集成测试

测试了主模块之后，有几种可能的测试顺序：

M1，M2，M3，M4，M5，M6，M7，M8，M9，M10，M11，M12

M1，M2，M5，M6，M10，M3，M7，M11，M4，M8，M12，M9

M1，M4，M8，M9，M11，M12，M3，M7，M2，M6，M10，M5

M1，M2，M6，M10，M4，M9，M5，M3，M7，M11，M8，M12

…

如果可以并行测试，那么还有一些可行的顺序。例如，测试了模块 M1 之后，一个测试员可以测试 M1 ~ M2 的组合，另一测试人员可以测试 M1 ~ M3 组合，第三个则可以测试 M1 ~ M4 组合。通常并没有最理想的测试次序，但下面两点是应该考虑的：

① 如果程序中有关键性的部分，比如模块 M7 是一个关键模块，设计测试次序就应该尽早地

把这些模块加入进程序。所谓“关键部分”可能是一个复杂的模块,一个具有新算法的模块,或怀疑容易出错的模块等。

② 设计测试次序时,要让输入/输出模块尽早地加入测试。

桩模块中的一部分是把它们的输入、输出信息送到终端或打印机上。当接受程序输入的模块加入时,测试用例的形式和最终程序接受的输入形式相同,测试用例的描述就大大简化了。同样,当执行程序输出功能的模块加入后,桩模块中可能不再需要放进写测试用例结果的代码。因此,如果模块 M10 和 M9 是输入、输出模块,并且 M7 执行一些关键性的功能,那么自顶向下的增量集成测试顺序最好是:

M1,M2,M6,M10,M4,M9,M3,M7,M5,M11,M8,M12

第6次增量测试后,程序中间形式如图6-5所示。

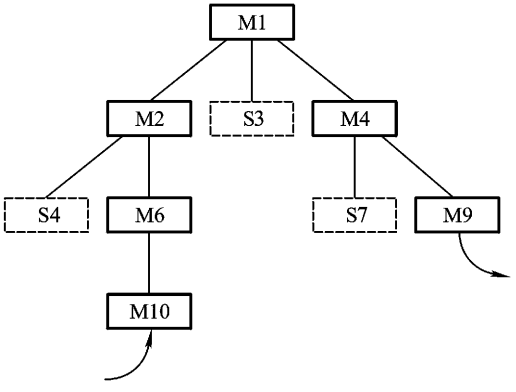


图6-5 自顶向下测试的中间状态

一旦达到图6-5这样的中间状态,测试用例的描述和结果检验都比较方便了。另一个好处就是程序工作轮廓比较清楚,换句话说,虽然程序“内部”是通过桩模块来模拟的,但却有了能执行实际输入/输出操作的程序形式。较早得到工作轮廓,可以较早地发现由于人为因素导致的故障或问题,可以对将来的用户表演程序,也可以作为程序总体设计正确的证据。对于一些人,还可起到精神鼓励作用。

另一方面,自顶向下测试也有一些严重的缺陷。比如,假设当前的测试状态如图6-5所示,下一步是用模块 M8 取代桩模块 S_7,那么就应当设计关于 M8 的测试用例集合。然而,测试用例实际上是程序输入给模块 M10 的数据。这样,就存在着一些问题:第一,由于介于 M10 与 M8 之间的一些模块(M6、M2、M1 和 M4),可能发现用模块 M10 的输入去测试模块 M8 中一些情况是不可能的。第二,由于模块 M8 与程序测试数据入口(M10)间的“距离”,即使可能测试模块 M8 中的每一种情况,要确定用什么样的数据输给 M10 才能测试模块 M8 中的这些情况,也是一件非常困难的脑力劳动。第三,由于一个测试的显示输出可能由远离被测模块的模块给出,在显示输出与被测模块之间建立联系也许是很困难的,有时甚至是不可能的。考虑在图6-5中加入模块 M5,每种测试用例的结果都由检验模块 M9 的输出来所决定,但由于中间模块的作用,也许很难推断模块 M5 的真实输出(也就是回送到 M2 的数据)。

关于自顶向下集成测试,可能还有两个更深入的问题。其一,有时以为这种测试能够与程序

的设计交错进行。例如,如果一个人正在设计图6-3所示的程序,当最上面两层已设计完毕,进入下一层设计时可能以为模块M1~M4能够进行编码和测试了,但这通常是很不明智的做法。程序设计是一个迭代过程,当设计低一层的程序结构时,可能要对上一层的设计进行必要的修改。如果发现这时上一层已经编过码了,并且已测试过了,就很可能会取消本来应该进行的修改工作。其二,实际中常常一个模块还没有彻底测试完又去测试另一个模块了。出现这种情况一方面是由于在桩模块中放入测试数据的困难性,另一方面是由于上层模块给下层模块提供了资源(例如,打开了某些文件)。从图6-3中可以看到测试模块M1可能需要多种形式的桩模块。实际上,因为测试M1要做大量的工作,如果等到模块M10加入到程序中之后,再来测试模块M1,这时测试用例的设计就容易多了。

总之,自顶向下集成测试可以自然地做到逐步求精,让测试人员看到系统的雏形,有助于对程序的主要控制和决策模块进行检验,增强测试人员的信心。不足之处在于:测试较高层模块时,由于低层处理用桩模块替代,不能反映真实情况,重要数据不能及时回送到上层模块,观察和解释测试输出往往比较困难。

为了解决这个问题,可以采用以下几种办法:

① 把某些模块测试推迟到用真实模块替代桩模块之后进行。

② 开发出能模拟真实模块的桩模块。

③ 采用自底向上集成测试方法。

第一种方法实际上是非增式集成测试方法,这种方法使故障难于定位和纠正,并且失去了在组装模块时进行一些特定测试的可能性。第二种方法无疑会大大增加开销;第三种方法是一种比较切实可行的方法。

2. 自底向上增式集成测试

自底向上增式集成测试是从软件结构的最下层模块开始测试的,测试较高层模块时,所需的下层模块功能都已具备,所以不再需要桩模块。

自底向上增式集成测试的步骤为:

① 把低层模块组织成实现某个子功能的模块群。

② 开发一个测试驱动模块,控制测试数据的输入和测试结果的输出。

③ 对每个模块群进行测试。

④ 删除测试使用的驱动模块,用较高层模块把模块群组织成完成更大功能的新模块群。

从第一步开始循环执行上述各步骤,直至整个程序构造完毕。

自底向上集成测试对最下层模块测试之后,同样也没有什么"最好的"方法来选择下一个要测试的模块,选择下一个待测模块的惟一原则是:所有下层的模块(即它能调用的模块)必须事先都被测试过。

对于图6-3所示的程序,自底向上增式集成测试的第一步顺序地或并行地测试M5、M7、M9、M10、M11和M12中的部分模块或全部模块。为此,每一模块都需要专门的驱动模块(参看图6-6),这个驱动模块可以接受测试输入,可以调用正在测试的模块,并且可以显示结果或将实际输出与期望输出进行比较。与桩模块不同,驱动模块可以反复调用正在测试的模块,所以并不需要多种形式的驱动模块。大多数情况下,设计驱动模块要比设计桩模块容易些。

如前面所述,影响测试顺序的因素是模块的关键性质。如果可以确定模块 M4 和 M6 是关键的,那么自底向上增式集成测试的中间状态可能如图 6 – 7 所示。下一步或许要测试 M5,然后把 M2 与前面已测试过的模块 M5、M6 和 M10 结合起来,再对 M2 进行测试。

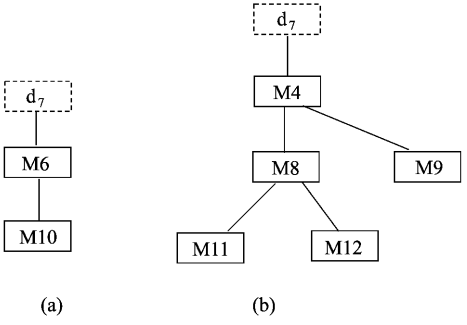


图 6 – 7 自底向上测试的中间状态

自底向上集成测试方法不需要桩模块,测试用例的设计亦相对比较简单,也不存在还没把前面的模块完全测试完又开始测试另一模块的问题。如果关键模块是在结构图的底部,自底向上测试具有一定的优越性。但在测试初期不能呈现出被测系统的轮廓。实际上,直到最后一个模块(模块 M1)加入时才具有整体形象,才能形成完整的程序。

对于大多数情况来说,自底向上集成测试和自顶向下集成测试正好相反;自顶向下集成测试的优点正是自底向上测试的缺点,而自顶向下集成测试的缺点却是自底向上集成测试的优点。

表 6 – 1 总结了自顶向下集成测试与自底向上集成测试方法的优缺点。每个方法的第一个优点可能是决定性因素,但因无法证明在典型的程序中故障经常发生在顶层还是底层,最保险的方法是对于被测试的程序,权衡表 6 – 1 中的各个因素。由于自顶向下集成测试中的第四个缺点会引起严重后果,并且我们可以用测试工具编写驱动模块(仍要桩模块)。因此自底向上集成测试方法更为优越。

表 6 – 1 自顶向下增式集成测试与自底向下的增式集成测试比较

集成方法	优点	缺点
自顶向下集成测试	① 主要故障发生在程序的顶端时,有利于查出故障 ② 一旦加入 I/O 功能,测试用例易于形成 ③ 初期的程序轮廓可以让人们看到程序的功能,增强信心	① 需要桩模块 ② 在 I/O 功能加入以前,很难描述测试用例 ③ 很难观察测试输出 ④ 使人想推迟完成某些模块的测试
自底向上集成测试	① 主要故障发生在程序的底端时,有利于查出故障 ② 测试用例生成容易 ③ 观察测试结果容易	① 需要驱动程序 ② 在加入最后一个模块之前,程序不能作为一个整体存在

自底向上的增式集成在最后一个模块加上之前程序不完整,自顶向下增式集成则在早期就有了程序的轮廓版本,但其涉及的桩模块较多,测试费用较高。它们各有优劣,一种方案的长处是另一种方案的短处。一种有效的选择是基于风险优选模块集成次序,混合采用自底向上和自顶向下的集成测试方法。例如,与高风险功能有关的模块可以较早进入集成测试,而与低风险功能有关的模块可以较晚进行测试。尽管人们倾向于先做容易的事情,但该混合方案提出的建议却正相反。

6.2 系统测试

为检测开发的软件在实际使用环境下,是否能够真正满足用户的需求,是否存在发生故障的隐患,必须对开发的软件进行规范的系统测试。也就是说,开发的软件只是实际投入使用系统的一个组成部分,因此还需要检测它与系统中其他部分能否协调地工作,这就是系统测试的任务。系统测试实际上是针对系统中各个组成部分进行的综合性检验,很接近日常测试实践。比如,在购买二手车时要进行系统测试,在订购在线网络服务时要进行系统测试等。这些大家所熟悉的测试形式的共同模式,就是根据人们的预期目标来评估产品,而不是根据规格说明或标准。因此,系统测试目标不是要找出软件故障,而是要证明系统的性能。比如,确定系统是否满足性能需求 确定系统是否满足可靠性要求等。

系统测试很困难,没有一套通用的方法。因此,系统测试需要真正的创造性。事实上,设计一套完备的系统测试用例,要比设计系统或程序需要更多的创造性,智慧和经验。

系统测试由若干个不同的测试类型组成,每一种测试都有一个特定的目标,然而,所有的测试都要充分地运行系统,验证系统各部分能否协调地工作并完成指定的功能。下面来简单介绍几类常用的系统测试。

6.2.1 性能测试

很多程序都有其特殊的性能或效率目标要求,说明在一定工作负荷和格局分配条件下,响应时间及处理速度等特性,例如传输的最长时间限制、传输的错误率、计算的精度、记录的精度、响应的时限和恢复时限等。

性能测试检测软件运行时的性能,为记录软件的运行性能常常需要在系统中安装必要的测量仪表或是为度量性能而设置的软件,即需要其他软硬件的配套支持。目前已有许多性能测试支持工具。

6.2.2 强度测试

强度测试检查系统能力的最高实际限度,即软件在一些超负荷情况下的运行情况。强度测试涉及时间因素,可用来测试那些负载不定的、或交互式的、实时的以及过程控制等程序。例如,测验一个打字员能否在一分钟内打出 60 个字符。如果航空控制系统可以跟踪管辖区内最多 200 架飞机,那么强度测试(模拟)检测 200 架飞机同时出现时系统会如何处理。实际上还可能有第 201 架飞机进入管辖区,所以,强度测试还要检测系统对这种意外情况的反映。再如,如果操作系统可以支持多达 15 道程序作业的运行,则强度测试是让 15 道作业同时运行起来。如果

分时系统最多可支持64个终端,那就让它处理64个终端用户同时调用系统的情况。实际工作中,一旦系统发生了故障,操作员立刻恢复系统时就会遇到这种情况。对过程控制系统的强度测试是让所有被控过程同时发出信号,对电话交换系统的强度测试是让它应付同时打来的大批电话。

强度测试中有些是实际使用中可能遇到的情况,有些在实际使用中不可能发生,但这并不意味着强度测试没有用。如果不可能遇到的测试情况发现了软件故障,那就是说,在实际的低强度情况下,类似的故障也可能发生。

6.2.3 安全性测试

安全性测试的目的在于检查系统对非法侵入的防范能力,验证安装在系统内的保护机构是否确实能够对系统进行保护,使之不受各种非常的干扰。安全性测试设法设计出一些测试用例,试图突破系统的安全保密措施。例如,① 想方设法截取或破译口令;② 编制专门软件破坏系统的保护机制;③ 故意导致系统失败,企图趁恢复之机非法进入;④ 试图通过浏览非保密数据,推导出所需的信息等,以检验系统是否有安全保密的漏洞。理论上讲,只要有足够的时间和资源,没有不可进入的系统。因此系统安全设计的准则是,使非法入侵的代价超过被保护信息的价值,此时非法入侵者已无利可图。

6.2.4 恢复测试

像操作系统,数据库管理系统以及远程处理程序等这样的程序,常常有系统恢复的目标,说明在软件出现故障、硬件失效或数据出错时,整个系统应如何恢复正常工作。恢复测试的主要目的是检查系统的容错能力,可以采取各种人工干预方式,比如将一些软件故障故意注入到操作系统中,制造通讯线路上的干扰,引用数据库中无效的指针等,使软件出错而不能正常工作,进而检验系统的恢复能力。如果系统本身能够自动地进行恢复,则应检查重新初始化、数据恢复和重新启动等机制是否正确。对于人工干预的恢复系统,还需估计平均修复时间,确定其是否在可接受的范围内等。

6.2.5 安装测试

安装软件系统时,用户可能会有很多种选择,比如分配并装入文件和程序库,设置适当的硬件配置,将程序和程序联系起来。因此,对安装过程进行测试也是系统测试的一个组成部分。安装测试的目的就是找出那些在安装过程中出现的错误,而不是软件故障。

6.2.6 可靠性测试

所有测试都以改善软件的最终可靠性为目的。但是,如果系统需求规格说明中有可靠性要求,就需要进行可靠性测试。通常使用以下几个指标来度量系统的可靠性:平均无故障时间是否超过规定的时限;因故障而停机的时间在一年中不应超过多少时间等。可靠性指标很难测试。比如,Bell系统的TSPS变换系统要求,每40年内因故障而停机的时间不能多于2小时。我们不知道有什么办法能在几个月,甚至几年内来测试这样一个指标。然而,如果可靠性指标是指平均无故障时间,如平均无故障时间为20小时,或运行出现的故障数目,如系统投入运行后不能出现

多于 12 个软件故障,那就可以用软件可靠性模型来评估这些指标。

6.2.7 配置测试

操作系统、数据库管理系统以及信息交换系统等,都是在许多硬件配合支持下工作的。如何保证软件在其设计和连接的硬件上正常工作,这是配置测试的工作目标。配置测试是用各种硬件和软件平台以及不同设置检查软件操作的过程,以保证测试的软件可以使用尽可能多的硬件组合。然而,现实世界中,各种型号的 CPU、打印机、显示器、网卡、调制解调器、扫描仪、数码相机、外围设备以及来自成千上万家公司的数百种计算机小产品——全都可以连到 PC 机上,并且每天都会有新的计算机设备问世,因此,不可能每种情况都测试到。

例如,假定要测试运行于 Microsoft Windows 的新软件游戏。该游戏画面丰富,具有多种音效,允许多个用户通过电话线进行对抗比赛,而且还可以打印游戏细节进行策划等。当考虑用各种图形卡、声卡、调制解调器和打印机进行配置测试时,Windows 的添加硬件向导允许用户从 26 种类型的硬件中选择,每一种硬件还有各种生产厂商和型号。这还只是 Windows 内置驱动支持的型号,还有许多自行提供安装盘的硬件型号。如果进行完整、全面的配置测试,检查所有可能的制造者和型号组合,就会面临巨大的工作量。市场上大约有 336 种显卡、210 种声卡、1 500 种调制解调器、1 200 种打印机。测试组合的数目是 $336 \times 210 \times 1\ 500 \times 1\ 200$,总计上亿种。

如果没有时间和计划测试所有的配置,就需要把成千上万种可能的配置缩减到可以接受的范围——即测试的目标。要测试哪些配置并没有一个定式,但在计划配置测试时一般采用的过程如下:

① 确定所需的硬件类型。应用程序要打印吗?如果是,就需要测试打印机。如果应用程序要发出声音,就需要测试声卡。如果是照片或者图形处理程序,还可能需要测试扫描仪和数码照相机。仔细查看软件的特性组合,确保测试全面彻底。

② 确定哪些硬件型号和驱动程序可以使用。

③ 确定可能的硬件特性、模式和选项。彩色打印机可以设置不同的打印模式,可以打印黑白照片或文字,也可以打印彩色文档。显卡有不同的色彩设置和屏幕分辨率。每一种设备都有选项设置,软件没有必要全部支持,但许多游戏要求最小颜色数和显示分辨率,如果配置低于该要求,游戏就不能正常运行。

④ 将硬件配置缩减到可以控制的范围内。

⑤ 明确使用硬件配置的软件特性。不必在每种配置中完全测试软件,只需测试那些与硬件相关的特性即可。例如,如果要测试写字板之类的字处理程序,因文件打印与打开和保存无关,设计好一个文档后,只需设法在选好的每一种打印机配置中打印该文档,而不必在每一种配置中都测试打开和保存特性。

⑥ 设计在每种配置中要执行的测试用例。

⑦ 反复测试直到对结果满意为止。

准备开始配置测试时,就应考虑那些与软件关系最为密切的配置。比如,对图像要求很高的计算机游戏应多加注意视频和声音部分。传真或通信软件则应测试多种调制解调器和网络配置。贺卡软件容易受打印问题的困扰,可以测试最流行的或者问世 5 年之内的打印机,还可以决定 75% 的测试针对激光打印机,而 25% 的测试针对喷墨打印机。不同的软件工程项目,可能有

不同的选择标准,但考虑的问题大致相同。

6.2.8 可用性测试

随着计算机的普及,用户的要求越来越高。可用性测试检测用户使用软件是否满意,具体体现为操作是否方便?用户界面是否友好?用户找到他们想要的东西是否容易?浏览菜单是否方便等。如果开发的软件难以理解、不易使用、运行缓慢或者用户指责软件不正确,这就是可用性测试的失败。可用性测试目的是让软件适合于用户的实际工作风格,而不是强迫用户的工作风格适应软件。

由于用户花费在软件界面上的时间比开发或测试人员多得多,看上去不重要的方面的影响就会变得越来越大,甚至会掩盖产品最有用的方面,所以许多软件公司花费大量的时间和费用来探索软件用户界面的最佳设计方式。优秀的用户界面应该包括下面7个要素。如果用户界面不符合这些要素,那么,软件就有缺陷了。

1. 符合标准和规范

最重要的用户界面要素是软件符合现行行业的标准和规范。这些标准和规范一般由软件可用性专家开发,经过大量正式的测试得出的方便用户使用的规则。如果软件严格遵守这些规则,优秀用户界面的其他要素自然就具备。

由于开发小组可能要对标准和规范有所提高,或者规则不能完全适用于软件,所以不会完全遵守这些规则。在这种情况下,需要真正注意可用性问题。

2. 直观性

1975年,第一台个人计算机微型仪器遥测系统MITS Altak8800问世了。它的用户界面如图6-8所示,除了开关和指示灯,一无所有,使用特别不直观。

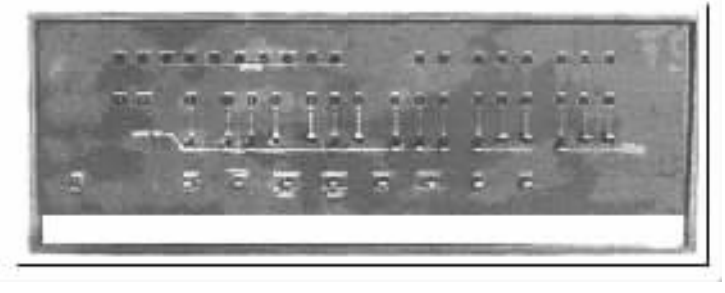


图6-8 MITS Altak8800的用户界面

今天,人们对软件的要求高多了。

当测试用户界面时,考虑以下问题,衡量软件的直观程度:

- 用户界面是否整洁,所需功能或者期待的响应是否明显。
- 用户界面布局是否合理,用户可以轻松自如地从一个功能转到另一个功能吗,任何时刻都可以决定放弃或者返回、退出吗,菜单或者窗口是否深藏不露。
- 有多余功能吗,是否有太多的特性把工作复杂化了,是否感到信息太庞杂。
- 如果其他所有努力失败,帮助系统真能起作用吗。

3. 一致性

一致性是一个关键属性。图 6 – 9 给出了两个 Windows 应用程序在 Find 上不一致的例子。在记事本程序中,Find 命令通过 Search 菜单或者按 F3 键访问,而在与其非常类似的写字板程序中,Find 命令通过 Edit 菜单或者按 Ctrl + F 组合键访问。

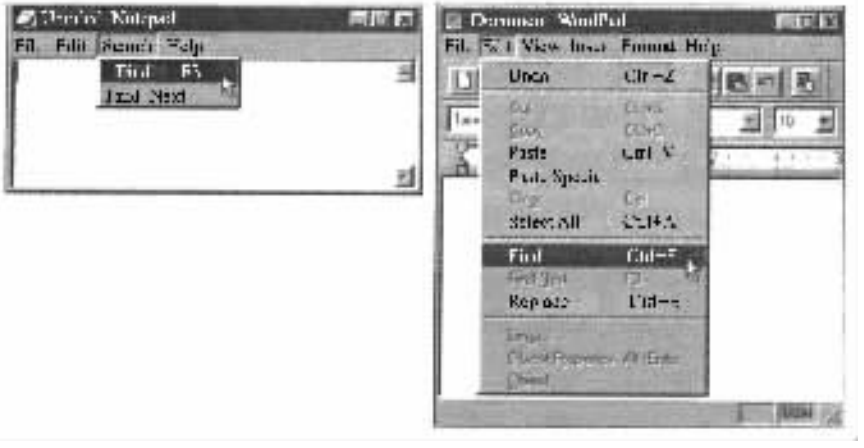


图 6 – 9 两个 Windows 应用程序在 Find 上不一致

像这样的不一致会使用户从一个程序转向另一个程序时感到不习惯,同一个程序中的不一致就更糟糕。在进行一致性检测时,应注意以下几点:

- 快捷键和菜单选项。在 Windows 中,按 F1 键几乎总可以得到帮助信息。
- 术语和命令。整个软件是否使用同样的术语,例如,Find 是否一直叫 Find,是否有时叫 Search?
- 按钮位置和等价键。大家是否注意到对话框有 OK 按钮和 Cancel 按钮时,OK 按钮总是在上方或者左方,而 Cancel 按钮总是在下方或右方?同样的原因,Cancel 按钮的等价键通常是 Esc,而 OK 按钮的等价键通常是 Enter。这些按钮和等价键应该保持一致。

4. 灵活性

用户喜欢做一些选择,但是不能太多。

5. 舒适性

软件使用起来应该舒适,如何鉴别软件是否舒适?下面一些方面可以用来鉴别软件的舒适程度:

- 恰当。软件外观和感觉应该与所做的工作和使用者相符。带有丰富用户界面的趣味贺卡程序不应该用来显示泄露技术机密的错误提示信息。相反,太空游戏则可以不管这些规则。
- 错误处理。程序应该在用户执行严重错误操作之前提出警告,并且允许用户恢复由于错误操作而丢失的数据。
- 性能。快不见得是好事。不少程序的错误提示信息一闪而过,无法看清。如果操作缓慢,至少应该向用户反馈操作持续时间,并且显示它正在工作,没有停滞。

6. 正确性

正确性测试就是测试用户界面是否做了它该做的事,应特别注意下面的情况:

- 市场定位偏差。有没有多余的或者遗漏的功能,或者某些功能执行了与市场宣传材料不相符的操作?不要拿软件与说明书比,而要与销售材料比。
- 所见即所得。用户界面所说的就是实际得到的吗?比如,单击 Save 按钮时,屏幕上的文档与存入磁盘的完全一样吗,从磁盘读出时与原文档相同吗?

7. 实用性

优秀用户界面的最后一个要素是实用性。这里所说的实用性不是指软件本身是否实用,而是指具体特性是否实用。

总之,不要让可用性测试的模糊性和主观性妨碍测试工作。

下面给出了一些用户界面错误的例子:

- 输入无合法性检查和值域检查,允许用户输入错误的数据类型。
- 界面中的信息不能及时更新,不能正确反映数据的状态,甚至可能对用户产生误导。如参数设置对话框中的预设值。

一些低效的用户界面例子有:

- 表达不清楚或过于模糊的信息提示。
- 要求用户输入多余的、系统可以自己得到的数据。如安装后用户需要修改某些配置文件。
- 为了达到某个设置或对话框,用户必须做许多冗余操作。如对话框嵌套层次太多。
- 不能记忆用户的设置或操作习惯,用户每次进入都需要重新设置初始环境。
- 不经用户确认就可对系统或数据进行重大修改。

6.2.9 兼容性测试

随着用户对各厂商各类型程序之间共享数据能力的要求加强,检查软件是否能够与其他软件正确合作变得越来越重要了。软件兼容性测试检测软件之间能否正确地交互和共享信息,其目标是保证软件按照用户期望的方式进行交互,是用其他软件检查软件操作的过程。

交互可以在运行于同一台计算机上的两个程序之间进行,也可以在相隔数千公里,通过因特网连接的两个程序之间进行,交互还可以简化为在软盘上保存数据,然后拿到其他房间的计算机上运行。

如果要对新软件进行兼容性测试,就需要回答以下几个问题:

- 软件要求与哪种操作系统、Web 浏览器和应用软件保持兼容,如果要测试的软件是一个平台,那么设计要求什么样的应用程序能在它上面运行。
- 应该遵守哪种定义软件之间交互的标准或者规范。
- 软件使用何种数据与其他平台和软件进行交互和共享信息。

1. 向前和向后兼容

关于兼容性测试的两个常用术语是向前兼容和向后兼容。向后兼容是指可以使用软件的以前版本。向前兼容是指可以使用软件的未来版本。并非所有软件都要求向前兼容或向后兼容。

软件测试人员应该为检测向前兼容和向后兼容提供相应的测试输入。

2. 不同版本之间的兼容性

测试平台和应用软件多个版本之间是否能够正常工作可能是一项十分艰巨的任务。比如要

测试一个流行操作系统的新版本,程序员修复了大量的软件故障,改善了性能,并在代码中增加了许多新的特性。当前操作系统上可能有上百万个程序。新操作系统的要求与它们百分之百的兼容。

由于不可能在一个操作系统上测试所有的软件程序,因此需要决定哪些是最重要的,是必须测试的,决定的原则可以是:

- 流行程度。利用销售记录选择前 100 或 1 000 个最流行的程序。
- 年头。应该选择 3 年以内的程序和版本。
- 类型。把软件分为画图、字处理、财务、数据库、通信等类型。从每一种类型中选择一个软件进行测试。
- 生产厂商。根据开发软件的公司来选择软件。

上面是关于新操作系统平台的兼容性测试的。测试新的应用程序也是一样,需要决定在哪个平台上测试软件,以及和什么样的应用程序一起测试。

3. 标准和规范

适用于软件平台的标准和规范有两个级别:高级标准和低级标准。

高级标准是产品普遍应遵守的,例如软件能在 Windows 或者 Linux 操作系统上运行吗?是 Internet 应用程序吗?如果是,运行于哪种浏览器上?每一个问题都关系到平台,如果某个应用程序声称与某平台兼容,就必须遵守关于该平台的标准和规范。

例如,Microsoft Windows 认证徽标就是一个例子。为了得到这个徽标,软件必须通过独立测试实验室的兼容性测试,其目的就是确保软件在 Windows 操作系统上能够平稳可靠地运行。

认证徽标对软件有以下几点要求:

- 支持三键以上的鼠标。
- 支持在 C 和 D 以外的磁盘上安装。
- 支持长文件名。

这些看上去是一些平常的简单要求,但这仅仅是长达 77 页文档中的 3 项而已。

低级标准是对产品开发细节的描述,从某种意义来说,低级标准比高级标准更重要。假如创建了一个运行在 Windows 之上的程序,但它与其他 Windows 软件在外观和感觉上有所不同,那么它就不会获得 Microsoft Windows 认证徽标。如果该软件是一个图形程序,可以把文件保存为.pict 文件格式,但程序不符合.pict 文件的标准,用户就无法在其他程序中查看该文件。软件与标准不兼容,很可能成为一个短命的产品。

同样,通信协议、编程语言语法以及程序员用于共享信息的任何形式都必须符合公开的标准和规范。因此,高级标准和低级标准都很重要,都需要测试以保证其兼容性。

4. 数据共享兼容性

在应用程序之间共享数据增强了软件的功能。支持并遵守公开的标准,允许用户与其他软件轻松地传输数据,这样的程序便是一个兼容性好的产品。

人们最熟悉的数据共享方式是读写磁盘文件,但文件的数据格式必须符合标准,才能在两台计算机上保持兼容。剪切、复制和粘贴也是程序间常见的一种数据共享方式。在这种情况下,传输通过称为剪贴板的程序实现。图 6-10 说明了这个传输过程。剪贴板设计来存放各种不同的数据类型。Windows 中常见的数据类型有文本、图片和声音等,这些数据类型可以有各种格式,

例如,文本可以是普通文字、HTML 等格式。如果对某程序进行兼容性测试,就要确认其数据可以利用剪贴板与其他程序相互复制。这个特性实在太常用了,以致人们想不起来其背后还有大量代码来保证其正常工作。但是对于保证所有对象能够正确链接、嵌入和数据交换的测试却是一个挑战。

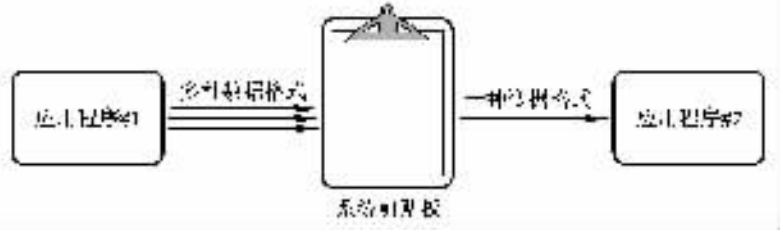


图 6-10 剪贴板用于存放各种不同的数据类型

6.2.10 文档资料测试

现在软件文档变得越来越大,有时甚至需要投入比开发软件本身还要大的时间和精力。软件测试不只限于测试软件,保证文档的正确性也是软件测试的职责。

(1) 可归于软件文档的内容

软件文档占到了整个软件产品的一大部分。以下是一些可归于软件文档的部分。

- 包装文字、标签和不干胶条。
- 市场宣传材料、广告以及其他插页。文档可能包含软件的屏幕抓图、特性清单、系统要求和版权信息等。
- 授权/注册登记表。
- 最终用户许可协议。
- 安装和设置指导。
- 用户手册。
- 联机帮助。
- 样例、示例和模板。
- 错误提示信息。

从用户的角度看,它们都是软件的一部分。

(2) 软件文档对软件整体质量的影响

对于严肃对待文档资料的用户而言,这些文档信息必须正确。如果联机帮助遗漏了一个重要条目,安装指导中存在错误的操作步骤,或者出现了显眼的拼写错误,用户就认为软件有缺陷。因此,所有这些文档都应该得到很好的检测。一个好的软件文档能从以下 3 方面提高软件产品的整体质量:

- 提高可用性。可用性大都与软件文档有关。
- 提高可靠性。可靠性是指软件平稳运行的程度。
- 降低支持费用。用户有麻烦或者遇到意外情况时,就会向公司请求帮助。好的文档能够通过恰当的解释和引导帮助用户克服困难,尽可能预防这种情况发生。

文档测试分为两个等级。如果文档是非代码,例如用户手册或者包装盒,测试可以视为技术编辑或校对。如果文档和代码紧密结合在一起,例如超级链接的联机手册等,就需要进行动态测试。表 6 – 2 列出了用做文档测试基础的简化的检查清单。

表 6 – 2 文档测试检查清单

检查项目	考虑的问题
用户	文档对同一级别的用户,难度合适吗
术语	术语适用于用户吗?用法一致吗?是否标准?需要定义吗?所有术语可以正确索引和交叉引用吗?公司的首字母缩写和术语完全相同吗
内容和主题	主题合适吗?有遗漏的主题吗?某个特性从产品中去掉,是否通知了手册的作者
	正确性
事实	所有信息真实而且技术正确吗?检查由于过期产品说明书和销售人员夸大事实而导致的错误
执行	仔细阅读文字,耐心补充遗漏的内容将执行结果与文档描述进行比较
图表和屏幕抓图	检查图表的准确度和精确度,其图像来源和图像本身对吗?图表标题对吗?确保屏幕抓图不是来源于已经改变的预发行版
样例和示例	像用户那样使用样例

(3) 文档开发与软件开发的不同

• 文档常常得不到足够的重视和预算。一般认为软件工程是第一位的,其他就不那么重要了。实质上,人们购买的是软件产品,除此之外就像位和字节那样无关紧要。如果负责测试软件中的一个领域,一定要为文档测试做出预算。

• 软件文档编写者可能对软件不甚了解。正如不必让会计师测试电子表格程序一样,文档编写者也不必是软件特性方面的专家。文档测试最重要的是,指出代码中难以使用或者难以理解的地方,以便他们能在文档中得到更好的解释。

• 印刷文档资料可能要花不少的时间,可能是几周,也可能是几个月。然而,软件可以立即发布到因特网或者光盘上。由于这个时间差,软件产品的文档需要在软件完成之前完稿。如果在这个时候改变了软件功能或者发现了软件故障,那么文档就无法反映最新的改进。解决这个问题的一个方法是找一个好的开发模式,使文档保持到最后一刻发布或联机发布尽可能多的软件文档。

由软件文档编写者、插图设计人员和索引编者等以各种形式创建的软件文档在开发和测试工作量上很容易超过实际的软件开发。

6.2.11 网站测试

因特网 Web 页是一个非常时尚的主题。实际上,因特网网页就是由文字、图形、声音和超级链接组成的文档。网络用户通过单击具有超级链接的文字和图形在网页间浏览,搜索查看找到

的信息。

网站测试囊括许多领域,包括配置测试,兼容性测试、可用性测试、文档测试等。当然,黑盒测试、白盒测试、静态和动态测试可能都要用上。

图 6 – 11 具有众多可测试特性的典型网页

图 6 – 11 给出了一个相当直观而且典型的中国软件网站的屏幕抓图。包含了多种可能的网页特性,部分特性包括:

- 不同大小、字体和颜色的文字。
- 图形和照片。
- 超级链接文字和图形。
- 不断变化的广告。
- 下拉式选择框。
- 用户输入数据的地方。

下面一些特性使网站更加复杂:

- 可自定义的布局,允许用户更改信息在屏幕上的位置。
- 可自定义的内容,允许用户选择想看的新闻和信息。
- 动态下拉式选择框。
- 动态变化的文字。
- 与不同网络浏览器、浏览器版本以及硬件和软件平台的兼容性。

网站测试是一项艰巨的任务,如何进行网站测试呢?最容易的起点是把网页或者整个网站当做一个黑盒子。

大多数网页比较简单,仅由文字、图形、链接以及少量表单组成。一般来说网页测试包括以下几方面内容:

(1) 文字测试

网页文字可以看做是软件文档,可以用测试文档的方法进行测试,检查用户等级、术语、内容、准确度——特别是可能过期的信息。

(2) 链接测试

链接是 Web 页的一个主要特征,它是在页面之间进行切换和指导用户去一些不知道地址的页面的主要手段。链接测试可分为 3 个方面。

- 测试所有链接是否按指示的那样确实链接到了该链接的页面。
- 测试所链接的页面是否存在。
- 保证网站上没有孤立的页面。所谓孤立页面是指没有链接指向该页面。孤立页面不能通过超级链接访问,因为网页制作者忘记把它挂接上,这需要将网页清单与 Web 服务器上的实际网页进行简单的覆盖分析,确定测试的是否是全部的网页,即既没有遗漏的,也没有多余的网页。

(3) 图形测试

一个 Web 页的图形可以包括图片、动画、边框、颜色、字体、背景、按钮等。图形测试的内容有:

- 确保图形有明确的用途,图片或动画不能胡乱地堆放在一起,以免浪费传输时间。
- 图片的大小和质量也是一个很重要的因素,一般采用 JPG 或 GIF 压缩。图片尺寸应小,但又能清楚地说明某件事情。
- 检测是否所有图形都正确载入和显示了。

(4) 表单测试

表单是指网页上用于输入和选择信息的文本框、列表框和其他域。表单测试检测域的大小是否正确,数据接收是否正确,可选域是否真正可选等一系列内容。

但是要真正找出重要的软件缺陷,需要对网站的系统结构和编程语言有一定的了解。

(5) 动态内容测试

动态内容是指根据当前条件发生变化的文字和图形——例如,日期、时间、用户喜好或者具体用户操作等。大多数动态内容编程在网站服务器上进行,要求具有 Web 服务器的访问权限才能查看源代码。

(6) 数据库测试

在 Web 应用中,数据库起着十分重要的作用,数据库为 Web 应用系统的管理、运行、查询和实现用户对数据存储的请求等提供空间。在使用了数据库的 Web 应用系统中,一般可能出现两种故障,一是数据一致性故障,二是输出故障。前者主要是由于用户提交的表单信息不正确而引起的,后者主要是由于网络速度或程序设计等问题引起的,针对这两种情况,应分别进行测试。

(7) 服务器性能和加载测试

流行网站每天可能要接受数百万次点击,每一次点击都要从网站的服务器上下载数据到浏览器的计算机。如果要测试系统的性能和加载速度,就必须找到一种方法来模拟数百万个连接和下载。

(8) 安全性测试

金融、医疗和其他包含个人数据的网站风险特别大,需要进行网站的安全性测试,主要涉及区域有:

• 现在网站基本采用先注册,后登陆的方式。因此,必须测试有效和无效的用户名和密码,检测是否可以不登陆而直接浏览某个页面等。

• 检测网页是否有超时的限制,即用户登陆在一定时间内(例如 10 分钟)没有点击任何页面,是否需要重新登陆才能正常使用。

• 当使用安全套接字时,检测加密是否正确,信息是否完整。

• 服务器端的脚本常常构成安全漏洞,这些漏洞又常常被黑客利用。所以,还要检测在服务器端放置和编辑脚本等问题。

测试一个网站,需要考虑可能会影响网站操作和外观的硬件和软件配置,网页是运用配置测试和兼容性测试的一个极好例子,遵守和测试一些基本规则也有助于使网站更加易用。

简单地单击所有链接,验证其正确性就会花去大量时间,再加上测试网站的基本功能,进行配置和兼容性测试,设法模拟数千甚至数百万个用户来测试性能和加载速度,任务太艰巨了。好在现在有一些免费测试工具,可以自动检查网站并测试其浏览器兼容性、性能问题、破碎超级链接、HTML 标准符合程度和拼写错误等。

小结

本章主要讲述了自顶向下增式集成测试与自底向上增式集成测试方法,并比较了两种方法的优缺点。实际应用中,针对具体问题,选择自顶向下增式集成测试或自底向上增式集成测试,或者采用自顶向下增式集成测试和自底向上增式集成测试相结合的方法进行测试。

系统测试是针对系统中各个组成部分进行的综合性检验,没有一套可以通用的方法。因此,系统测试需要真正的创造性。

常用的系统测试有 :性能测试、强度测试、安全性测试、恢复测试、可靠性测试、安装测试、配置测试、可用性测试、兼容性测试、文档资料测试、网站测试。

第 6 章习题

1. 自底向上增式集成测试与自顶向下增式集成测试的主要区别是什么,各有什么优缺点?
2. 常用的系统测试有哪些?
3. 如何辨别发现的软件故障是普遍的问题还是配置问题?
4. 所有的软件必须进行某种程度的兼容性测试吗?
5. 兼容性可以具有不同程度的符合标准吗?
6. 如果要对产品的数据文件格式进行兼容性测试,应如何进行测试?
7. 列举出熟悉产品中用户界面设计不当或不一致的例子。
8. 既然用户界面没有明确的对与错,那么如何进行可用性测试?
9. Widows 画图程序帮助索引有 200 多个条目,是否要测试每一个条目以检测能否到达正确的帮助主题,假如有 10 000 个索引条目又如何?
10. 好的文档以哪 3 种方式提高软件产品的整体质量?
11. 利用黑盒测试方法可以测试网页的哪些基本元素?
12. 进行网站测试时要考虑哪些要领?

第 7 章　验证测试和确认测试

20 世纪 70 年代以后，人们逐渐认识到，要想开发出高质量的软件产品，必须重视开发前期的质量检验工作，必须找出一种合适的方法来检验软件开发各阶段的质量。验证就是对诸如需求规格说明，设计规格说明和代码之类的产品进行评估和审查的过程，有时被称为人工测试，可应用到开发早期一切可以被评审的事物上，以确保该阶段的产品正是所期望的。实践证明，验证是一条最可靠、最有效的质量改进之路。

确认是在开发过程中或结束时进行的评估系统或组成部分的过程，目的是判断系统是否满足规定的要求，包括实际软件或仿真模型的运行，是‘基于计算机的测试”。

验证和确认是为捕获不同类型的软件故障而设置的过滤器，它们相互补充，以保证最终软件产品的正确性、完善性和一致性。

本章重点：

- 验证的基本方法
- 验证活动
- 确认策略
- 确认活动

7.1　验证的基本方法

验证是对软件产品进行人工检查或评审的过程。验证的基本方法有‘软件审查、走查、伙伴检查等，它们有许多相似之处，但也不尽相同，一般认为软件审查是最正规的方式。表 7－1 给出了各种验证方法的基本特征。

表 7－1　验证方法的基本特征

项目	软件审查	走查	伙伴检查
主持人	非该软件的编制人员	任何人	没有
参与人员	3～6 人小组	多一些人	1～2 人
准备	有	只有主持人	无
数据收集	有	不要求	无
输出报告	有	不要求	口头评论
优点	有效	能使更多人熟悉产品	费用低
缺点	短期成本高	查出的故障较少	查出的故障较少

7.1.1 软件审查

正式评审、技术评审以及软件审查会是软件审查的几种不同说法。软件审查以会议的形式进行,利用集体的智慧查找软件产品中存在的问题,从而保证软件产品的质量。软件审查的对象可以是任何重要的工作产品,例如,需求分析、概要设计、详细设计等阶段的成果以及源程序代码、测试计划和测试用例等。

(1) 软件审查的目标

- 收集数据,发现软件故障。
- 交流信息。

(2) 软件审查的基本要素

- 确定问题。软件审查的目标是为了找出软件中存在的问题——不仅是出错的地方,还包括遗漏或多余的地方。全部的批评应该直指被审材料,而不是创建者或编写者。
- 遵守规则。软件审查要遵守一套固定的规则,比如设定要审查的工作量,花费的时间等,其重要性在于使参审人员了解自己的作用及预期的目标,这有助于使审查工作进展顺利。
- 准备。每一个参审人员都应了解自己的责任和义务,为审查做好准备,并积极参与审查工作。审查过程中找出的问题大部分是在准备期间发现的,而不是在审查会上发现的。
- 编写报告。审查小组必须做出总结审查结果的书面报告,诸如发现了多少问题,在哪里发现的,以便于开发小组使用。

(3) 软件审查的输入

- 被审查的文档。
- 有关的原始资料。
- 审查单。

(4) 软件审查的输出

- 审查汇总/报告。
- 有关故障类型的数据。

(5) 软件审查的步骤

① 制定计划

在软件产品具备阶段审查的条件后,可以着手制定审查工作计划。首先要确定审查会的主持人,为客观公正,主持人不应该是被审查软件的开发人员或被审查文档的编写者。主持人负责检查软件产品是否确已具备了进行审查的条件,确定参加审查的其他人员,负责组织审查会,评估审查工作的结果。参审人员包括开发人员、测试人员和‘局外人”,以 3 至 6 人为宜,这样可以使会议参加者听到更多的不同声音。为了能准确地完成评审,主持人应在会议之前,将有关的材料提供给参审人员,要求他们在会前熟悉这些材料,做好参加会议的准备,从而有可能在会上提出不同的看法和解释。

② 准备

在准备软件审查工作时,开发人员收集相关的审查资料,其他参审人员则认真阅读和研究所提供的资料,充分了解被审查的软件,记录发现的问题。在这些问题中,有些可能是明显的错误,有些可能是不好理解的部分,这些问题都将在审查会上提出来,以求得进一步的分析和讨论。

③ 审查会

召开审查会是软件审查工作的中心环节,主要包括:

• 主持人了解会议准备情况,如果他认为准备工作不够充分,可以决定推迟审查会召开的时间。

• 仔细阅读并记录所发现的问题是审查会的主要工作。在审查会上,请审查员逐段朗读被审资料,参审人员可随时提出问题。经验表明:有许多故障是在叙述过程中被审查员自己发现。换句话说,大声朗读被审资料就是一种相当有效的故障检测方法。

• 主持人决定这次审查的结论:符合要求、返工或需要再次审查等。

审查会的目的是要找出并记录被审查软件产品存在的问题,主持人要确保讨论有效地进行,并使所有参审人员集中精力发现故障,而不是修复故障,修复故障是在会议之后由开发人员完成。

④ 返工

审查返工自然是在审查会以后由开发者完成的。这项工作通常只是把记录在"审查会发现问题报告"中的错误改正过来。

⑤ 终审

终审是由主持人完成的最后一项审查活动。在返工完成以后,主持人要检验所有需要改正的地方是否确已改正。

(6) 软件审查的功效

如果审查工作开展得很成功,通常可以发现一些重大的软件缺陷。它们好比是一张捕虫网,在开始阶段可以捕捉到一些大一点的虫子。诚然,小虫子可能穿过这张网,但它们将在下一阶段被网孔更小的网捕住。

除了发现问题这个主要功效之外,坚持软件审查还有几个有益的间接效果:

• 交流。审查有助于加强彼此间的交流。不能单纯从所发现的故障衡量审查的成功与否,审查将改进软件开发过程。人们通过交流和讨论,能更多地了解对他们工作有益的东西,例如,通过加入审查小组,测试人员可以了解产品的基本信息,缺少经验的测试人员可以向有经验的测试人员学习,审查的交流价值不容忽视。

• 质量。审查有助于提高工作积极性,提高产品的正确性。从某种意义上说,它利用了人们的荣誉感,甚至是窘迫感。试想一下,如果人们知道他们的工作要被审查,他们肯定将做得更加出色。

• 小组同志化。如果审查正确进行,就会成为软件测试人员和开发人员相互领略对方技艺的最佳时机,并且能够更好地了解他人的工作及要求。

总之,软件审查提供了一个把大家聚在一起讨论同一个项目问题的最好机会。

7.1.2 走查

走查不像软件审查那么正式,准备工作一般由主持者负责,参加人员只是简单地参加会议,在会议前不需要做更多的准备工作。

走查的目标是:熟悉材料、发现软件故障。

走查的主要元素有:

- 定期的会议,只有主持人必须事先准备。
- 2~7人,软件开发人员或材料编写者。
- 主持人通常是软件开发者。

走查的输入是被检查的材料以及可使用的标准等。

走查的输出是走查报告。

因为主持人是软件开发者,其他参加人员没有多少负担,使得走查工作的覆盖面可以比软件审查大一些,从而给更多的人提供了一个熟悉了解材料的机会。有时,走查的目的不是为了发现软件故障,而只是为了熟悉材料,进行交流。软件可能是"继承"过来的,我们并不知道它到底是什么,于是组织走查,把一些人关在屋子里,对它进行逐页通读。如果能发现一些问题当然好,但主要目的是使自己更加熟悉材料,了解产品。

由于走查的主持人是软件开发者,使得检查的客观性受到一定影响,这是走查工作的不足。

7.1.3 伙伴检查

伙伴检查通常是在编写代码的程序员和充当审查者的其他一、两个程序员或测试人员之间进行,这个小团体只是在一起审查代码,寻找问题和失误。即便如此,聚集起来讨论问题也能找出一些软件缺陷。发现产品中的缺陷而不是人的错误,伙伴检查不失为一种极好的训练。

7.1.4 建议

任何形式的检查,哪怕只是简单地将文档交给别人,要求他们仔细阅读,只要是由编写者以外的其他人来做,只要是以发现错误为目的,就总能发现几个自己无法发现的错误,总比不做检查要强多了。

一般来说,建议进行正式的评审和审查。尽管审查的短期费用比较高,但只要处理得当,收益总会超过成本。实践表明,软件审查是一种最有效、最切实可行的验证方法。一个对照实验表明,软件审查方式用于检查程序时,可以发现30%到70%的逻辑设计错误和编码错误。IBM公司的代码审查会查错效率高达80%。

由于软件审查很费时而且要求精力高度集中,一般用来处理相对较小的材料。有时我们可能说"这部分代码是最关键的,是系统的心脏,因此可以挑选几个人,对这部分代码的每一行都进行仔细的审查。对重要文件、样本进行审查也很重要,可以估计文件的质量以及未审查部分包含故障的数量。有时我们可能说"这部分代码不太重要,用不着软件审查会。对于这种情况,可以进行不太正规的评审、走查、伙伴检查或这些形式的变体。

对关键文档应进行完全验证,但对于大型软件产品来说,像所有的测试活动一样,完全验证是不切实际的,这时,就需要进行风险评估、根据实际情况做出取舍。进行验证测试时,通常采取的顺序是:工作产品规模小的在前,大的在后,简要的在先,详细的在后,验证成本低的在前,高的在后,潜在收益大的在先,小的在后。这意味着如果资源或进度的限制不能进行某些方面的验证,那么省去验证的顺序与上面验证的顺序正好相反。

提高验证效益的一个关键因素是,必须保证不会在错误的时间对错误的东西进行验证,所验证的应该与确认测试以及最终交付的是同一件东西。

7.2 验证活动

验证活动是测试生存周期中的一个阶段,包括需求验证、功能设计验证、详细设计验证和代码验证。在每个验证活动中,测试的目的都是为了发现尽可能多的故障,测试人员应积极参与软件审查和走查工作,并开展验证工作。

在每一类的验证活动中,都要考虑以下问题:

- 使用的验证方法(审查、走查、伙伴检查等)。
- 产品中要验证的和不要验证的范围。
- 没有验证的部分所承担的风险。
- 需要优先进行验证的范围。
- 验证设计的资源、进度、工具和责任等。

7.2.1 审查单

审查单是验证测试的重要工具,尤其是对于像软件审查这一类比较正式的验证方法。针对各种类型的验证活动都有相应的审查单,例如,需求验证审查单,功能设计验证审查单,详细设计验证审查单,代码验证审查单、一般文档验证审查单等。对于一切要进行检查的项目都可以开发相应的审查单。

但是,开发部门与测试部门常常使用不同的审查单,开发部门使用的审查单常常注重产品的可维护性,以及编码标准和规范之类的东西,而测试部门使用的审查单则更注重产品的可靠性和可用性等方面的特性。

进行验证测试时,最重要的是利用一般的审查单,针对特殊目的和具体项目开发属于自己的审查单,这些审查单应该有明确的目的,并能充分反应测试机构在验证测试方面现有的成熟水平。审查单是验证测试的重要工具,为了最大限度地发挥验证的作用,它们应得到妥善的管理,及时的更新和发展,以确保不同项目和不同人员开展的验证工作得以延续。同时,审查单还是培训的重要手段,可记录验证工作前进的步伐。

7.2.2 需求验证

1. 需求分析

用户需要什么?开发的项目试图为用户提供什么?问题的答案正是需求阶段所要说明的。由于很难完全理解用户的需要,因此需求开发十分困难。一直以来需求阶段都是软件链中最为薄弱的一环。

需求可以是正式编制的文档,也可以是表示用户要求的非正式的通信,需求可能是明确的,也可能是含糊的。需求阶段的测试目标就是要保证在开展下一步工作之前能完全了解用户的需要。

需求可能还会涉及一些重要和基本的问题,像安全性、可用性、可维护性以及一些不明确或不好解释的性能。当从测试的角度出发检测需求时,所有含糊不清楚的地方都要应被问清楚。一个“好的性能”究竟是什么,是2秒的响应时间,还是24小时的响应时间,一般来说,需求越量

化,其最终系统就越成功,测试也就越简单。如果没有对需求规格说明达成一致的意见,要进行开发是很困难的,而要进行合理的测试则更困难。

需求规格说明的质量往往与编写者是否知道该如何撰写需求以及他们是否熟悉标准有关。需求规格说明的质量还是开发机构成熟度水平的重要标志之一。

2. 需求验证审查单

对需求进行验证测试,就是要找出那些可能是问题的地方,搞清楚它们是否完整、连贯、合理、可实现、可追踪,从测试的角度看可度量。

通用需求验证审查单一般包括以下几方面的内容:

(1) 完整性

每一条款必须详细说明所列问题的解决方案。

(2) 准确、清楚

每一条款只有一种解释并且容易理解。

(3) 前后一致

规格说明中的前后条款不能自相矛盾。

(4) 相关性

每一条款都与问题和其最终解决方案关联。

(5) 可测试性

项目开发及验收测试期间,能确定该项目是否得到满足。

(6) 可跟踪性

每一条款都能追踪到它的起源。

(7) 可行性

每一条款在规定的时间和开销下,在现有的人力和物力支持下,都可以实现。

通过对需求进行验证测试,可以为确认测试以及变更谈判打下良好的基础。同时,需求验证最有可能为软件开发节省成本,它可以检测出许多的缺陷。事实上,50% 以上的缺陷实际上都是在需求阶段引入的,如果这些不足进入开发周期的后期,那时再纠正的代价就太大了。

7.2.3 功能设计验证

1. 功能设计

设计是软件工程的技术核心,功能设计是将用户需求转化为一组外部接口的过程,其功能设计规格说明从功能角度对产品行为进行描述。

对功能设计进行验证测试,就是要确定用户需求是如何在功能设计中得以具体体现的,需求中的每一项都要在功能设计规格说明中得到体现,如果不是这样的话,它是被遗漏了还是被删掉了?

功能设计规格说明不完整是一种最常见的缺陷。优秀的审查或评审人员不仅能阅读摆在他面前的那些东西,而且还能不断地设想 如果由我来编写功能设计规格说明,我会在这部分之前包含什么内容,应该如何来描述它们,如何兼顾到最终的用户等。通过假想自己是功能设计文档的编写者,可以发现一些遗漏的错误,这些都是很重要的。

2. 功能设计验证审查单

一份通用的功能设计验证审查单常包含以下几方面的内容：

- 如果一个术语已被明确定义，则尽可能用定义代替术语。
- 如果对一个结构进行了描述，则试着勾画被描述结构的结构图。
- 如果指定了一项计算，则至少手工计算两个例子并作为范例包含在规格说明内。
- 查找确定性语句时，应根据需要一层层地进行搜索，直到满足计算机所需要的确定性。
- 注意含义不明确的词，如一些、有时、经常、通常、主要地、习惯上、大多数等。
- 查找功能清单，找出没有例子或例子雷同的功能，修改例子并包含在规格说明内。

7.2.4 详细设计验证

1. 详细设计

详细设计是将功能设计转化为详细的数据结构、数据流和算法的过程，它表明了软件产品具体是如何构造的。为体现不同的抽象层次，可能会有多个详细设计规格说明，如果可能，对它们中的每一个进行验证测试。

详细设计规格说明对测试人员来说十分宝贵。从测试的角度出发，对详细设计规格说明进行仔细的审查，可以利用审查获得的信息，了解产品将如何构造，系统将如何集成，可以使测试人员更加深入和全面地认识软件产品。

2. 详细设计验证审查单

使用详细设计验证审查单对详细设计规格说明进行验证测试，发现问题可以追踪到功能设计和需求阶段，看看所用的算法及组合方式是否合适。在寻找错误的同时考虑应该如何测试。例如，如果在详细设计中用到了一个表格，测试人员就可以问一些与测试相关的问题“表格有多大？如何填充表格？怎样充满表格？表格为空时将会发生什么？”等。这样的提问可能激发一些测试思路，设计出更多的基于详细设计的测试用例。

一个典型的详细设计验证审查单中包括：

- 设计文档是否包含了详细设计的步骤描述或说明。
- 是否有计算机系统的用户界面模型。
- 是否有计算机系统其他接口的模型或描述。
- 是否有被推荐的计算机系统高层功能模型。
- 主要实施及评估方案在详细设计规格说明中是否得到了反映。

IEEE/ANSI 在软件工程标准文档中提供了软件设计描述的推荐实践，包括信息、格式及材料组织方式的确定等。

7.2.5 代码验证

1. 编写代码

编写代码是将详细设计规格说明转换为代码的过程。许多公司喜欢以程序代码作为审查或走查工作的起点，也许它是一个很舒服的起点，但它肯定不是一个最有效的起点，因为当人们对代码进行走查或审查时，可能还得回去检查那些在代码之前完成的文档。针对一个糟糕的功能设计规格说明或错误的需求进行编码，已经浪费了许多宝贵的时间。

2. 代码验证测试

对代码进行验证测试可以采用审查、走查或伙伴检查的方式,但无论采用哪种验证方式,都应以测试的观点思考问题,看到一段代码,考虑一些新的测试,考虑那些由需求和功能设计规格说明不可能想到的测试,从而提高代码查错的效率。

代码验证测试主要包括以下几方面的内容:

- 将代码与详细设计规格说明进行比较。
- 对照特定编程语言的审查单检查代码。
- 使用静态分析工具对句法规则进行检查。
- 检查代码中的标识符是否与在数据字典和详细设计规格说明中的一致。
- 寻找边界条件、可能的性能瓶颈以及其他可能需要进行测试的内部条件。

3. 标准或规范

在代码验证测试时,测试人员还应检查代码编写是否符合某种标准或规范。如果代码可以正常运行,但编写不符合某些标准或规范,仍然认为软件有缺陷。这好比写的东西可以被人理解,但是不符合语言的语法或文法规则。坚持标准或规范的原因有:

- 可靠性。事实证明按照某种标准或规范编写的代码比不按标准编写的代码更可靠,软件故障更少。
- 可读性/维护性。符合标准和规范的代码易于阅读、理解和维护。
- 移植性。代码经常需要在不同的硬件上运行,或者使用不同的编译器编译。如果代码符合标准,移植到另一个平台轻而易举,甚至完全没有障碍。

一般而言,标准是必须遵守的规则——做什么和不做什么。规范是建议和推荐的最佳做法。标准没有例外情况,类似于结构严谨的法律公文,规范相对就松一些。是遵守国家标准、国际标准还是内部规范并不重要,重要的是开发小组在编写代码和正式审查验证中拥有标准和规范。

7.3 通用代码审查单

如果是正式地开展代码审查,就应该使用机构中现有的审查单,或在开始时使用通用代码审查单,然后在其基础上开发定制出适合某种目的或具体项目的审查单。下面是一张通用的代码审查单:

1. 数据引用错误

数据引用错误是指使用未经正确初始化或说明的变量、常量、数组、字符串或记录而导致的软件故障。检测数据引用错误,应考虑以下几方面的问题:

- 是否引用了未初始化或未说明的变量。查找遗漏与查找错误同样重要。
- 数组和字符串的下标是否整数,下标是否在维数定义的范围之内。
- 使用字符串时,是否超过了字符串的边界,使用数组时,是否出现‘差一个’这样的潜在错误。
- 是否在应该使用常量的地方使用了变量(例如在检查数组范围时)。
- 是否变量被赋予了不同类型的值(例如,无意中为整形变量赋予了一个浮点数值)。
- 是否为引用的指针分配了内存。
- 如果一个数据结构在多个函数或者子程序中被引用,每个引用对该数据结构的定义是否一致。

2. 数据声明错误

数据声明错误是指不正确地声明或使用了变量或常量。检测数据声明错误应考虑以下几方面内容：

- 变量是否都赋予了正确的长度、类型和存储类型，例如，本应声明为字符串的变量是否声明为字符数组了。
- 变量是否在声明的同时被初始化了，是否正确初始化并与其存储类型相匹配。
- 变量是否有相似的名称。这未必是缺陷，但可能是程序中出现名称混淆的信息。
- 是否存在声明过、但从未引用或者只引用过一次的变量吗。
- 模块中所有变量都显式声明了吗，如果没有，是否该变量为更高级别的模块所共享。

3. 计算错误

计算或者运算错误是基本的数学逻辑问题。检测计算错误应考虑以下几方面内容：

- 计算中是否使用了不同数据类型的变量，例如将整数与浮点数相加。
- 计算中是否使用了数据类型相同但长度不同的变量，例如将字与字节相加。
- 计算时是否了解或考虑了编译器对类型或长度不一变量的转换规则。
- 目标变量（保存计算结果的变量）的分配空间是否小于右边的表达式。
- 在数值计算过程中是否可能出现溢出。
- 除数/模是否可能为零。
- 对于算术运算，代码处理是否可能丢失精度。
- 变量的值是否会超过合理的范围，例如，概率的计算结果是否会小于 0% 或者大于 100%。
- 对于包含多个操作数的表达式，求值的次序是否混乱，运算优先级是否正确，是否需要加括号使其清晰。

4. 比较错误

比较和判断错误很可能是边界条件问题。检测比较错误应考虑以下几方面内容：

- 比较运算符正确吗？比较应该是小于还是小于或等于。
- 是否存在分数或者浮点数之间的比较，如果有，精度问题是否会影响比较结果。
- 每一个逻辑表达式是否都正确，求值次序是否有问题。
- 逻辑表达式的操作数是否逻辑值，是否将整型变量用于逻辑计算中。

5. 控制流错误

计算或者比较错误常常造成控制流错误。检测控制流错误应考虑以下几方面内容：

- 如果程序包含 begin…end 和 do…while 等语句组，结尾与其相应的语句组是否匹配。
- 程序、模块、子程序或循环能否终止，如果不能，是否可以接受。
- 是否有死循环，是否会过早地退出循环。
- 是否有这种可能，由于入口条件的原因，某个循环永远不会被执行到。
- 如果程序包含有多个分支，是否能执行到每个分支。
- 是否存在‘ 差一个 ’错误，导致意外地进入或退出循环。

6. 接口错误

接口错误主要是由于模块间不能正确地传递参数所造成的。检测接口错误应考虑以下几方面内容：

- 被调用模块接收的参数类型和数量与调用模块传送的参数是否匹配，次序是否正确。

- 如果某模块有多个入口点,引用的参数是否与当前入口点无关。
- 常量是否被当做形参传递。
- 子程序是否更改了只能作为输入的参数。
- 每个参数的使用单位与形参匹配吗,例如,度与弧度。

7. 输入/输出错误

输入/输出错误包括文件读取、接受键盘或鼠标输入出错,以及向打印机或屏幕等输出错误。检查输入/输出错误应考虑以下几方面内容:

- 如果文件被显式地声明,它们的属性是否正确。
- 是否处理了文件、外设不存在或未准备好等错误的情况。
- 所有文件在使用之前是否都被打开了。
- 软件是否以预期方式处理预计的错误。
- 是否检查了错误提示信息的准确性、正确性、语法和拼写错误。

8. 其他检查

其他检查定义了一些不适合放在其他类别的条目。其他检查应考虑以下几方面内容:

- 软件是否使用了其他外语,是否处理了扩展的 ASCII 字符。
- 软件是否要移植到其他编译器或 CPU 上,是否具有这样做的可能性。如果没有计划或测试,那么,移植性可能成为一个大难题。
- 是否考虑了兼容性,以使软件能够运行于不同的硬件配置,例如图形卡和声卡,不同的外设等。
- 程序编译是否产生‘ 警告 ’或者‘ 提示 ’信息。检查每一条信息,这可能意味着软件的有效性存在问题。
- 模块是否健壮,是否检查了输入的有效性,是否遗漏了某个函数。

利用验证测试可以尽早地发现和检测出一些软件故障,但还有许多故障是验证测试发现不了的。在大多数组织中,验证测试/确认测试发现的故障分配比例为 1/4,验证测试甚至更低。图7 – 1

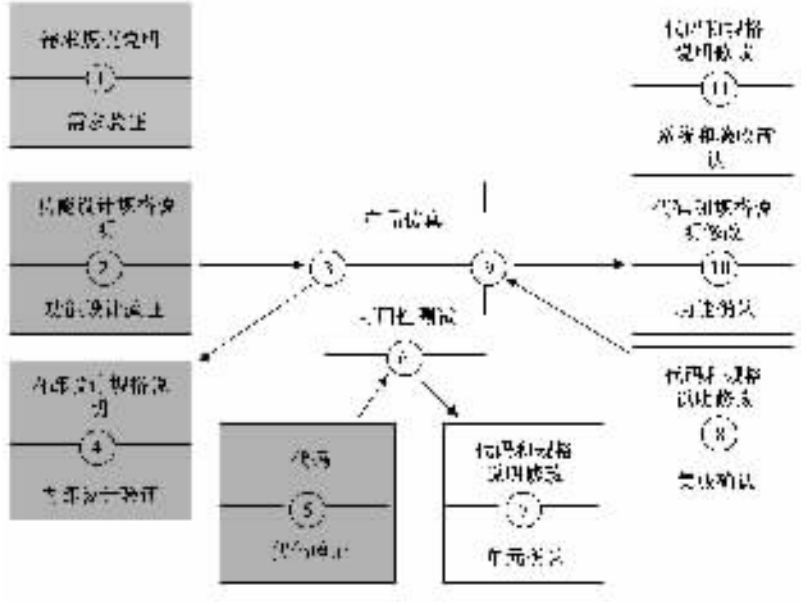


图 7 – 1 软件开发技术的 Dotted – U 模型

给出了一个软件开发技术的 Dotted – U 模型,清楚表明了验证测试和确认测试各自的适应范围。

7.4 确认测试

确认测试以需求规格说明中的规定作为检验尺度,在开发过程中或结束时,对系统或组成部分进行评估,确认开发的软件是否满足需求规格说明规定的所有功能和性能,文档资料是否完整、人机界面和其他方面(例如,可移植性、兼容性、故障恢复能力及可维护性等)是否令用户满意等。

实践中,如何来判断一个程序是否确实符合需求呢?办法有两个:

① 检测产品是否满足在需求规格说明中规定的用户需求。

② 检测产品的实际行为是否与功能设计规格说明中描述的一样。

7.4.1 确认任务

1. 确认

确认是指决定最后的软件产品是否正确无误。比如,开发的软件是否符合软件需求规格说明和用户要求,输出的信息是是用户想要的信息,在将来的实际使用环境中能否正确稳定地运行,是否存在隐患等,这自然包含了对它在功能、性能、接口以及限制条件等方面满足需求程度的评价。

如前所述,穷举测试是不现实的,任何程序的测试都是不完整的。测试的目标是从大量的输入数据中挑选出少数有代表性的测试数据,使得采用这些测试数据去检测被测程序,能够检测出尽可能多的软件故障,将这种不完整性的负面影响降到最低,这样,便可以通过有限的测试,发现尽可能多的故障。

2. 测试覆盖

测试覆盖用来衡量软件产品的被测程度。设计的测试用例在多大程度上覆盖了被测产品,测试用例涉及了被测产品的多少需求,检测了被测产品的多少功能,测试了被测产品的多少内部逻辑,这些都可以用测试覆盖来衡量。

测试覆盖分为 :需求覆盖、功能覆盖和逻辑覆盖,即用测试用例集对被测产品的需求覆盖、功能覆盖和逻辑覆盖程度来衡量产品的被测程度,为此,必须确保在每一层次上都有足够的测试。

IEEE/ANSI 已清楚地阐述了需求覆盖和功能覆盖的重要性。事实上,逻辑覆盖也很重要的,因为:

① 逻辑覆盖可以间接地提高功能覆盖。

② 对逻辑路径的测试是必要的,而逻辑路径一般无法从外部功能看出来。比如,实现一个数学函数可以采用完全不同的算法。

逻辑覆盖包括 :语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖和路径覆盖等。最常用、最简单的形式是语句覆盖,即覆盖被测程序中语句的百分比。例如,“ 80% 覆盖 ”意味着程序中 80% 的语句被检测过。在一个编译程序测试的项目中发现,如果语句覆盖为 75% 时,发现了 x 个故障,那么当将语句覆盖从 75% 提高到 85% 时,还会发现 x 个新故障。同样,当语句覆盖从 85% 提高到 90% ,又会发现 2x ~ 3x 个新故障。一般来说,发现新故障的概率与没有被覆盖的代码总量

成反比。越接近100%的覆盖率,测试的就越有可能是以前没有被覆盖的地方,也就越有可能发现更多的软件缺陷。

7.4.2 确认测试策略

黑盒测试和白盒测试是两种基本的确认测试方法。

黑盒测试基于功能设计规格说明,在对软件内部结构一无所知的情况下设计测试用例,进行测试。但是,黑盒测试不能检测已经实施但在功能设计规格说明中没有被描述的功能,与之相关的故障用黑盒测试方法也发现不了。

实施白盒测试则必须了解软件的内部结构,是基于详细设计规格说明或代码的测试。但是,白盒测试无法检测那些在功能设计规格说明中进行了描述,但在详细规格说明中没有描述或代码没有实现的功能。

(1) 确认测试策略

第4章和第5章详细介绍了各种常用的黑盒测试方法和白盒测试方法,每种方法各有优缺点。利用这些测试方法,同时考虑到需求覆盖、功能覆盖和逻辑覆盖等方面的要求,可以制定下面的3种确认测试策略:

- 基于需求的测试。基于需求的测试必须采用黑盒测试方法,在不知道详细设计规格说明或代码的情况下对用户需求进行测试。基于需求的测试根据功能设计规格说明设计测试用例。
- 基于功能的测试。基于功能的测试应该采用黑盒测试方法,根据功能设计规格说明,采用等价类划分、边界值分析和故障猜测等方法设计测试用例。
- 基于内部的测试。基于内部的测试只能采用白盒测试方法,但可采用功能设计规格说明制订测试计划。一旦采用白盒测试,便可通过一系列的技术确保系统的内部各部分获得充分的测试并且达到足够的逻辑覆盖。

无论哪种测试策略,都应满足结果明确和可重复性。

测试用例的一个重要组成部分是预期的输出或结果。如果事先没有定义预期的结果,很容易将实际结果视为正确的结果。

测试用例应该是可重复的,即同一测试用例在相同的软件/硬件配置上运行时应能产生同样的结果,但可重复性并不总是可以实现的,例如软件/硬件在处理异步过程时,就很难实现可重复性。

(2) 实施确认测试时应考虑的基本原则

- 测试是为了发现故障,而不是为了显示故障不存在。
- 测试最困难的问题之一是不知道何时停止测试。
- 避免使用未经计划、不能重复使用并且用后即扔的测试用例。
- 测试用例的一个重要组成部分是预期输出或结果,仔细比较每一次测试执行的实际结果和预期结果。
- 测试用例必须考虑有效、无效、预期和非预期的输入条件。
- 经验较少的测试人员倾向于只从输入角度设计测试用例,经验丰富的测试人员能够生成预期输出所要求的输入。

7.4.3 确认测试活动

确认测试是基于计算机测试的过程,确认测试活动可以分为 :低层测试和高层测试。

1. 低层测试

低层测试涉及对程序的各个单元或模块进行测试,包括单元测试和集成测试。一次一个模块或将它们组合起来进行测试,这需要详细了解程序的内部结构。

(1) 单元测试

单元测试针对程序的各个模块,依据详细设计规格说明设计测试用例,进行测试。单元测试应对模块内所有重要的逻辑路径进行测试,以便发现模块内部或模块接口存在的故障。单元测试多采用白盒测试方法,多个模块可以并行地进行测试。

(2) 集成测试

集成测试是将多个模块组合起来进行测试的过程,集成测试的主要目的是发现模块接口之间存在的故障。集成测试有几个层次,可以集成和测试一个程序的不同模块,一个子系统的不同程序、一个系统的不同子系统或网络中的不同系统等。

在非增式的集成测试中,将所有的模块集成起来组合成一个完整的程序,然后再进行测试。实践证明,这是一个效率很低的方法,由于故障可能牵扯到任何模块,调试非常困难。

在增量集成测试中,涉及模块接口之间不匹配的故障能够较早地被发现,纠正故障相对容易,测试也更详尽些,并且测试新模块时会对以前测试过的模块提供进一步的功能和逻辑覆盖。

(3) 综合测试策略

Myers 提出了使用各种测试方法的综合策略:

- 在任何情况下都必须使用边界值分析方法。经验表明,用这种方法设计出的测试用例发现软件故障的能力最强。
- 必要时使用等价类划分方法补充一些测试用例。
- 用故障推测法再追加一些测试用例。
- 对照程序逻辑,检查已设计测试用例的逻辑覆盖程度。如果没有达到要求的覆盖标准,应当再补充一些测试用例。
- 如果程序的功能说明中含有输入条件的组合情况,则一开始就可以选用因果图或决策表法。

2. 高层测试

高层测试涉及对整个软件产品的测试。为保证测试的客观性,测试最好在开发机构之外由独立的测试机构进行。高层测试的形式包括:

- 可用性测试
- 功能测试
- 系统测试
- 验收测试。

(1) 可用性测试

可用性测试基于需求规格说明,属于系统测试的一部分。在第 6 章的系统测试中已重点介绍过可用性测试,其在开发周期中的重要性已将它提升到了更加显著的位置,以至在图 7 – 1 的

Dotted – U 模型中将其单独作为一部分。

可用性测试检测软件用户界面与实际用户需求之间的差异,涉及用户对产品的反应,但它与 Beta 测试不同。尽管可用性测试也涉及用户,但它应该尽早在开发周期中实施,尽早让真正的用户参与进来,甚至屏幕还只是画在纸上时就可参加进来。

可用性测试是一种确认测试活动而不是验证测试活动,它需要一个真正的用户通过仿真或实际执行最终产品,它检测产品的表现而不是对功能进行评估。

可以测试的可用性特征包括:

- 可访问性 用户是否可以较为容易地进入、浏览和退出。
- 反应性 用户是否可以在需要的时候,以一种明确的方式做他们想做的事情。
- 有效性 用户是否可以以最少的步骤和时间做他们想做的事情。
- 可理解性 用户是否理解产品结构、帮助系统和文档资料。

(2) 功能测试

功能测试以功能规格说明为依据,以检测程序的功能规格说明与实际行为之间的差异为目标。如果出现了偏差,可能是程序有问题,也可能是规格说明有问题,还可能是程序和规格说明都有问题。既然功能测试是基于功能规格说明的,必须对功能规格说明进行仔细的分析,对功能进行无遗漏、无冗余、无矛盾的描述。所有黑盒方法都适用于功能测试。

功能测试在软件提交给用户之前由测试小组进行,只要产品具备足够的功能便可以执行某些测试或在完成单元或集成测试之后,就可以开始功能测试。

测试用例的执行将检测程序的一些功能。功能覆盖程度表示测试用例集对程序功能的覆盖程度,可用如表 7 – 2 所示的功能覆盖矩阵表示。功能覆盖矩阵很简单,是一个矩阵或表格,列出要测试的具体功能、测试的重点以及所用的测试用例。功能测试要求做到一定程度的功能覆盖。

表 7 – 2 功能覆盖矩阵

功能	重点	测试用例
…	…	…

功能测试可以分几步进行:

① 分解并分析功能设计规格说明。

② 将功能细分为若干类,对每一类列出详细的功能。

③ 对每一功能,确定其输入和输出。

④ 采用黑盒测试方法设计功能测试用例。

⑤ 开发功能覆盖矩阵。

⑥ 执行测试用例并度量被测程序的功能覆盖和逻辑覆盖程度。

⑦ 根据功能覆盖和逻辑覆盖的程度,补充一些测试用例。

(3) 系统测试

系统测试是检测程序或系统是否满足其需求规格说明所规定的需求和目标的过程。因为没有专门的方法可循,系统测试是一种最容易被人误解并且最难的测试活动。可以针对各种不同类型的系统测试构造测试用例。

系统测试可由需求覆盖来度量,测试用例的执行将检测软件的一些需求,需求覆盖程度表示测试用例集对软件需求的覆盖程度,可以用需求覆盖矩阵或需求跟踪矩阵表示。

系统测试由测试小组在用户获得产品之前进行。只要软件产品具备一定的功能或者在单元和集成测试完成之后就可以开始系统测试,也可以与功能测试并列进行。但一般将系统测试推迟到功能测试已表明预定的可靠性水平之后进行,最好在 40% 的功能测试完成之后再进行。

系统测试可以分以下几步进行:

① 分解并分析需求规格说明。

② 将需求划分成一系列的逻辑类,对每一逻辑类列出详细的需求清单。

③ 对于每一类系统测试:

(a) 对每一相关需求,确定输入和输出。

(b) 开发需求测试用例。

④ 开发需求覆盖矩阵。

⑤ 执行测试用例并度量需求覆盖和逻辑覆盖程度。

⑥ 根据需求覆盖和逻辑覆盖的程度,补充测试用例。

3. 验收测试

验收测试一般是在测试小组完成了可用性测试、功能测试和系统测试之后,软件产品向用户交付之前进行的最后一次质量检验活动,是将最终产品与最终用户需求进行比较的过程。回答开发的软件产品是否符合预期的各项要求,以及用户能否接受等问题。由于验收测试不只是检验软件某个方面的质量,而是要进行全面的质量检验,并且要决定软件是否合格,因此验收测试是一项严格的正规测试活动。事实上,开发人员不可能完全预见用户实际使用软件的情况。例如,用户可能错误地理解了命令,或提供了一些奇怪的数据组合,亦可能对开发者自认为清楚明了的输出信息迷惑不解等。因此,软件是否真正满足最终用户的要求,应由用户进行一系列验收测试。在验收测试中应考虑的问题包括 验收的标准、验收测试的步骤和方法、验收测试的组织、验收争端的解决等。

验收测试针对软件质量,主要检测以下几方面的内容:

• 可靠性。用户最为关心的是已开发的软件是否能够正确地、稳定地工作。有的软件验收协议规定,在整个软件项目验收期间,源程序中发现的故障数不得超过程序总量的万分之几,如果情况不是这样,可靠性要求就没有满足,其他质量特性不需再检验,软件被拒绝接受。

• 性能。其次考虑软件的性能指标。响应时间是用户十分关心的性能指标,例如,10 个用户同时使用软件时,响应时间为 1 s。吞吐量是另一个重要的性能指标,它以单位时间内软件产生的产品数量来衡量。这两项指标达到了要求,才能考虑是否通过验收,如果达不到要求,软件性能太差,即使可靠性满足了要求,软件也很可能被拒绝接受。

• 可理解性。由于可理解性不容易定量衡量,因此只能由用户作出判断。一般首先考虑,软件的各种文档是否齐全,其次,文档编写的内容是否遵循了指定的规范,是否必要的内容都已写入 ;最后,文字叙述是否流畅、易读。

• 可修改性。如同可读性一样,可修改性也无法严格定义。可修改性度量方法的指导思想是,软件的修改是费时的,而且容易引入新的故障,应尽可能避免修改软件本身。

因为验收测试关系到软件的命运,应对软件作出负责任的、符合实际情况的客观评价,这一

评价不应渗入人为因素,比如偏向于软件开发者或是偏向于用户一方,应尽可能去掉一些人为的模拟条件,或是去掉一些开发者的主观因素,使得验收测试出到真实、客观的结论。

由最终用户主持的验收测试可以是非正式测试,也可以按事先制定的计划,系统地进行正式的测试。事实上,验收测试常常要进行几个星期,甚至几个月。如果软件是为多个用户开发,那么由每个用户都实施正式的验收测试是不可能的。大多数软件产品的开发人员采用 Alpha 测试和 Beta 测试,以找出只有最终用户才能发现的软件故障。

7.4.4 累进测试和回归测试

累进测试是对软件进行测试以确定是否存在故障的过程。回归测试则是对软件进行测试以确定是否因故障修复而引入了新的故障。大多数的测试用例开始时都属于累进测试用例,最后成为软件产品的回归测试用例。回归测试不是一种新的测试活动,它是为检查因修复故障可能引入的新的故障而重新执行某些或所有测试用例的过程。可以为每一个测试活动进行回归测试,比如,单元测试、集成测试、可用性测试、功能测试、系统测试等。

7.4.5 测试执行

(1) 测试用例设计

设计测试用例时需考虑的测试执行方面的问题:

- 是采用许多小的测试用例还是采用几个大的测试用例。
- 测试用例间的相互依赖关系。
- 测试执行的主机环境。

(2) 测试用例执行

一个测试用例可能包含一个或多个分测试。如果测试用例发现了故障,可能会出现两种情况:

- 测试用例的执行可能被提前终止,这样一来,后续分测试的执行被屏蔽了。这意味着只有在修复故障或修改测试用例后,后续分测试才能被执行。
- 测试用例继续执行下去,就像没有出现故障一样。

具体出现哪种情况,视测试用例本身、测试环境和被测软件而定。

(3) 测试屏蔽

测试屏蔽可能发生在:

- 测试用例之间。如果测试用例 X 依赖测试用例 Y,那么 Y 引起故障时,X 的剩余部分可能就不能被执行了。
- 测试用例之内、分测试之间。在一个测试用例内,如果一个分测试引起了故障,后续的分测试可能就不能被执行了。
- 分测试之内、输入之间。在一个分测试内,如果一个输入引起故障,其余的输入可能就不能被测试了。

例如,在测试一个程序接口时,分测试调用软件过程之一 PROC,过程 PROC 以一定的顺序检查输入参数的有效性。当检测到一个无效参数时,过程可能中止或将一个出错码返回给调用者,提前结束执行,剩下的其他无效参数的检查都不能进行。一个有效输入如果引起了故障,也

可以暂时屏蔽其他的输入条件,直到故障被修复为止。

(4) 对测试用例设计的一些建议

- 对于使用有效输入的分测试,采用大的测试用例。
- 对于使用一个无效输入的分测试,每个分测试采用一个测试用例。
- 一个测试用例可以依赖另一个程序,但不应该依赖另一个测试用例。
- 执行测试用例时,主机系统的状态必须一致。
- 工作模式不应发生变化。这与每一个测试用例的可重复性有关,如果测试用例发现了故障,但工作模式却发生变化,故障的起因可能被掩盖起来,这时,要诊断原因是非常困难的。

(5) 进行确认测试时的建议

- 针对功能测试,系统地设计并开发基于功能的测试用例,针对可用性和系统测试,采用黑盒测试方法和基于需求的测试。
- 执行测试,并使用覆盖工具度量它们的逻辑覆盖程度。
- 评审并分析测试结果。
- 如果某一区域显示不成比例的大量故障,则应强化该区域的测试。

基于外部的测试应该由不知道被测程序内部结构的人员进行。掌握了内部结构会使测试出现偏向,避免的办法是请别人设计不同类型的测试,或者让同样的人在基于外部的测试进行过后,再设计基于内部的测试。

小结

验证测试是对诸如需求规格说明,设计规格说明和代码之类的产品进行评估、评审和审查的过程。确认测试则是在开发过程中或结束时,对系统或系统部分进行评估以确定其是否满足需求规格的过程。

验证活动是测试生存周期中的一个阶段,包括需求验证、功能设计验证、详细设计验证和代码验证。确认活动也是测试生存周期中的一个阶段,包括低层测试(单元测试及集成测试)和高层测试(可用性测试、功能测试、系统测试及验收测试)。

在经过了验证测试和确认测试后,应该分析:

- 通过验证发现了多少错误。
- 有多少是由验证发现的,又有多少是在后面的确认测试中发现的。
- 有多大比例的故障留到了测试结束,是由用户发现的。

需求验证最有可能为软件开发节省成本,它可以检测出许多的软件缺陷。如果这些不足进入开发周期的后期,那时再纠正的代价就要高多了。确认测试的结果有两种可能,一种是功能和性能指标满足软件需求说明的要求,用户可以接受;另一种是开发的软件不满足软件需求说明的要求,用户无法接受,项目进行到这个阶段才发现严重故障和偏差一般很难在预定的工期内改正,因此必须与用户协商,寻求一个妥善解决问题的方法。

总的来说,验证的效果要高于确认测试,它可以发现一些在确认测试过程中几乎不可能检测出来的故障。作为一种策略,应该加大验证测试的比例。

第 7 章习题

1. 正式审查由哪些关键要素组成？
2. 验证测试和确认测试的主要区别是什么，确认测试能否取代验证测试？
3. 进行验证测试的好处有哪些？
4. 除了更正式以外，经验测试与验证测试有什么主要的差别？
5. 简述确认测试和其他测试（如白盒测试、黑盒测试）之间的关系。
6. 确认测试活动包括哪几个重要的方面？

第 8 章　测试计划与测试文档

软件测试的目标是以尽可能少的成本,尽可能早地找出软件中尽可能多的故障,并保证其得到修复。事实上,高效率的测试是经过计划的,成功的测试需要有一定的方法。利用组织良好的测试计划、测试用例和测试报告,正确交流和制订测试工作是测试人员达到目标的保障。本章主要讨论与每一个测试活动有关的具体任务和可交付的文档。

本章重点:

- 测试计划
- 验证测试计划
- 确认测试计划
- 软件测试文档

8.1　测试计划

如果要测试一个大系统,可能需要编写出好几万个测试用例,然后执行这些测试用例并检验测试结果。在这个过程中,可能要涉及上百个模块,修改上千个故障,花上一年或更多的时间,有时还可能要雇佣几百个人进行测试。如果测试人员之间不能很好地交流计划测试的对象,需要使用的资源,进度安排等,那么整个项目就很难成功。事实上,高效率的测试是经过计划的,成功的测试需要有一定的方法,包括条例、结构、分析和度量等。利用组织良好的测试计划、测试用例和测试报告,正确交流和制订测试工作是测试人员达到目标的保障。没有组织、目标,任意随便的测试注定是要失败的。软件测试计划是测试小组与产品开发小组交流意图的主要方式。

不言而喻,计划是指导一个测试过程的决定性部分。好的测试计划应该包括以下几方面的内容:

- 目的。必须明确每一测试阶段的目的。
- 测试策略。测试策略描述测试小组用于测试整体和每个阶段的方法。
- 资源配置。计划资源要求是确定实现测试策略必备条件的过程。如果需要特殊的硬件设备,计划中就要给出相应的要求以及什么时候使用这些设备等说明。具体的资源要求取决于开发的项目、测试小组和公司等因素。
- 任务明确。任务分配明确,具体指出谁负责软件的哪些部分、哪些可测试特性,以确保软件的每一部分都有人测试,每一个测试人员都清楚自己的职责,而且有足够的信息开始设计测试用例。
- 进度安排。对每一个测试阶段,制定一个详细的进度安排表。测试进度安排可以为项目开发小组和项目管理员提供信息,以便更好地安排整个项目的进度。
- 风险。明确指出项目中潜在的问题或风险区域,在进度中给予充分的考虑。
- 停止测试的标准。必须给出判断每个测试阶段停止测试的标准。

• 测试用例库。测试计划过程决定用什么方法编写测试用例,在哪里保存测试用例以及如何使用和维护测试用例。

• 组装方式。包含若干程序或分系统的系统可能是逐次地组装在一起的,测试计划应确定组装的次序,确定是按自顶向下还是自底向上的增式集成方式进行测试,确定系统在各种组装下的功能特性以及桩模块或驱动模块的设计。

• 记录手段。必须明确一种方法,用来记录测试中所遇到的各种问题和取得的进展,包括找出容易出错的模块,估计相对于时间表的测试进展,可供使用的资源等。

• 工具。必须确定所需要的测试工具并制定出相应的计划:谁来开发或负责这些工具,如何使用这些工具以及什么时候使用等。

• 回归测试。回归测试的目的是确定故障修复是否对其他方面造成了影响。制定回归测试计划也是有必要的(如谁来进行,怎样进行,何时进行测试等)。

8.2 软件测试文档

软件测试文档用来描述要执行的测试及测试的结果。软件测试是一个很复杂的过程,涉及软件开发其他阶段的工作,对于提高软件质量,保证软件正常运行有着十分重要意义,因此必须把对测试的要求、过程及测试结果以正式的文档形式写下来。可以说,测试文档的编制是软件测试工作规范化的一个重要组成部分。

软件测试文档不只在测试阶段才开始考虑,它应在软件开发的需求分析阶段就开始着手编制,设计阶段的一些设计方案也应在测试文档中得到反映,以利于设计的检验。测试文档对于测试阶段的工作有着非常明显的指导作用和评价作用。即便在软件投入运行的维护阶段,常常也要进行再测试或回归测试,这时仍会用到软件测试文档。

1. 测试文档的类型

根据测试文档所起的作用,通常把测试文档分成两类,即测试计划和测试分析报告。测试计划详细规定测试的要求,包括测试的目的和内容、方法和步骤,以及测试的准则等。由于要测试的内容可能涉及到软件的需求和设计阶段的工作,因此必须及早开始制定测试计划,不应在开始测试时,才考虑测试计划。通常,测试计划的编写从需求分析阶段开始,直到软件设计阶段结束时才完成。

测试报告用来对测试结果进行分析说明,说明软件经过测试以后,结论性的意见如何,软件的能力如何,存在哪些缺陷和限制等,这些意见既是对软件质量的评价,又是决定该软件能否交付用户使用的依据。由于要反映测试工作的情况,自然应该在测试阶段编写。

《计算机软件测试文档编制规范》国家标准给出了更具体的测试文档编制建议,其中包括以下几个内容:

• 测试计划。描述测试活动的范围、方法、资源和进度,其中规定了被测试的对象,被测试的特性、应完成的测试任务、人员职责及风险等。

• 测试设计规格说明。详细描述测试方法,测试用例设计以及测试通过的准则等。

• 测试用例规格说明。描述测试用例涉及的输入、输出,对环境的要求,对测试规程的要求等。

- 测试步骤规格说明。规定了实施测试的具体步骤。
- 测试日志。日志是测试小组对测试过程所作的记录。
- 测试事件报告。报告说明测试中发生的一些重要事件。
- 测试总结报告。对测试活动所作的总结和结论。

前 4 个属于测试计划,后 3 个属于测试分析报告。

2. 测试文档的重要性

软件测试文档的重要性表现在以下几个方面:

- 验证需求的正确性。测试文档中规定了用以验证软件需求的测试条件,研究这些测试条件对弄清用户需求的意图十分有益。
- 检验测试资源。测试计划不仅要用文档的形式把测试过程规定下来,还应说明测试工作中必不可少的资源条件,进而检查是否具备了这些资源,即它的可用性如何。如果某个测试计划已经编写出来,但所需资源仍未落实,那就必须及早解决。
- 明确任务的风险。了解测试任务的风险有助于对潜伏的可能出现的问题事先作好思想上和物质上的准备。
- 开发测试用例。测试用例的好坏决定着测试工作的效率,选择合适的测试用例是作好测试工作的关键。在测试文档编制过程中,按规定的要求精心设计测试用例十分重要。
- 评价测试结果。测试文档包括测试用例,即若干个测试数据以及对应的预期结果,将测试结果与预期的结果进行比较,便可对已进行的测试给出评价意见。
- 回归测试。测试文档规定和说明的内容对维护阶段进行回归测试时,非常有用。
- 确定测试的有效性。完成测试以后,把测试结果写入文档,可为分析测试的有效性,甚至整个软件的可用性提供依据,同时还可以证实有关方面的结论。

8.3 主测试计划

(1) 目标

制定主测试计划的目标是 :规定测试活动的范围、方法、资源和进度,明确正在测试的项目,要执行的主要测试任务,每个任务的负责人以及与计划相关的风险等。制定主测试计划是要在上层文档中弄清全貌,要进行什么类型的测试,有多少验证测试工作,进行什么样的确认测试,需要什么样的整体策略等。

(2) 风险分析

风险分析是任何开发计划的一部分,它对于确定要进行哪些测试、测试的力度如何十分关键。风险是指任何威胁到项目目标成功实现的因素。每一个项目实施都有一定的风险,实际测试总要受时间和资源的限制,不可能对项目中的所有方面进行完备的测试。测试的一个基本的原则是对项目中最具风险的部分进行详细的测试,以确保最具破坏性的故障能够被发现。

因此,在测试的每个层次上确定基于风险的优先级问题非常重要。主测试计划风险管理要考虑的问题包括:

- 软件的大小和复杂性。
- 软件的关键性,关键软件的失效可能会影响到安全,甚至可能造成巨大的经济或社会

损失。

- 软件开发过程成熟度。
- 测试形式,进行全面测试还是部分测试。
- 测试人员、经验和组织方式等。

(3) 验证和确认测试计划大纲

测试计划的结果是某一种文档,但它只是制定详细测试计划过程的一个副产品。制定测试计划的主要目的是交流测试小组的意图、期望以及对将要执行任务的理解。重要的是计划过程,而不是生成文档。ANSI/IEEE 标准 829/1983 推荐了一种常用的软件测试文档格式,应该能够有效地交流测试工作的进展。

主测试计划可交付的文档之一是软件验证测试计划和确认测试计划,按照 IEEE/ANSI 标准 1012/1986 进行验证测试和确认测试可以为每一测试阶段提供综合性的评估,保证在软件生存周期内尽早地发现并排除故障,减少项目开发的风险提高软件质量的可靠性。

按照 IEEE/ANSI 标准 1012/1986,软件验证和确认测试计划大纲包括以下几方面的内容:

- 目的。
- 参考文件。
- 定义。
- 验证和确认测试概要——主进度表、资源概要、责任、工具、技术和方法等。
- 生存周期的验证和确认测试任务——生存周期中,每个阶段的验证和确认测试任务、输入和输出等。
- 软件验证和确认测试报告——对所有验证和确认测试报告的内容、格式和时间分配进行描述。
- 验证和确认测试管理——涉及测试管理的策略,应遵循的步骤、标准、惯例、协议等。

IEEE 1012/1986 标准包括图表和说明,可以帮助制定测试计划,对将要实施的验证和确认测试活动进行描述,对计划内容进行对照检查等。软件验证和确认测试计划附属于总的软性质量保证计划。

8.4 验证测试计划

验证测试活动包括:

- 需求验证。
- 功能设计验证。
- 详细设计验证。
- 代码验证。

因此,验证任务包括对各个验证活动制定测试计划并执行测试。

8.4.1 制定验证测试计划

(1) 制定验证测试考虑的问题

是否要对需求进行验证测试,要验证软件的全部需求吗?采用什么样的验证方法,是进行全

方位的正式软件审查,还是走查或伙伴检查,要对软件的哪些区域进行验证,只验证 80% 的代码可以吗,不验证的风险是什么,资源调度情况如何?

制定验证测试计划要考虑以下几方面的问题。

- 要进行的验证活动(需求验证、功能设计验证、详细设计验证、代码验证)。
- 采用的方法(审查、走查等)。
- 软件需要或不需要进行验证测试的区域。
- 不对软件某些区域进行验证测试的风险。
- 所需的资源、进度、责任、设备、工具等。

每个验证活动都要制定相应的验证测试计划,对采用的验证方法、软件中要验证和不要验证的区域、相关的风险、优先考虑的区域等做出详细的描述。

(2) 验证测试计划大纲

验证测试计划(大纲)包括以下几方面的内容:

- 测试计划标识。
- 验证活动。
- 要验证和不要验证的区域。
- 任务。
- 责任。
- 人员配备和培训。
- 进度安排。
- 风险和意外事故。
- 审批。

8.4.2 验证执行

验证执行可采用软件审查、走查、伙伴检查等方式进行。

(1) 审查报告

每次验证执行后应提交一份相应的审查报告说明审查了什么,谁在进行审查,进度提前了多少,出错率有多高,在什么地方发现了故障,故障的严重程度如何,故障修复后是否需要重新进行审查,审查是否成功,有多少返工等?

审查报告(大纲)可包括如下几方面的内容:

- 审查报告标识。
- 测试项目和版本。
- 参审人员。
- 被审查材料的规模。
- 审查小组的准备时间。
- 返工的工作量及返工结束的预期日期。
- 故障清单。
- 故障概要(分类整理的故障数量)。

(2) 验证测试报告

另一个与验证测试有关的可交付文档是验证测试报告。验证测试报告是对验证活动的概要说明。验证的目标是对所有与软件有关的文档进行验证测试,但最终验证了多少?出现了哪些需要解决的内部问题?哪些已经完成?哪些还没有完成?可以把验证测试报告当做一种执行概要,用于提高管理层对测试过程的认识,引起他们对有关问题的注意。每一验证活动都应有一个验证测试报告,目的是对这一验证活动的所有验证进行概括总结。

8.5 确认测试计划

(1) 确认测试活动

- 单元测试和集成测试。
- 可用性测试。
- 功能测试。
- 系统测试。
- 验收测试。

(2) 确认测试任务

- 将所有确认测试活动看做一个整体,制定主确认测试计划。
- 以确认测试活动为单位制定详细测试计划,开发测试用例,执行测试,评估测试,维护测试。

8.5.1 制定确认测试计划

(1) 制定确认测试计划时应考虑的几个问题

- 确认测试采用的方法。
- 测试执行所需的设备。
- 测试工具和支撑软件。
- 测试配置管理。
- 风险分析、预算、资源要求、进度安排、人员配备等。

确认测试准备采用什么样的测试方法,选用哪种测试自动化技术或测试工具,需要什么样的支撑软件和人员配备,如何进行配置管理,这些方面的考虑既适用于整体测试也适用于每个确认测试活动。

(2) 主确认测试计划大纲

主确认测试计划是为整个确认测试工作制定的,是对整个确认测试工作的概括,即确定具体的确认测试活动,每一确认活动需要的资源,每一确认活动粗略的进度安排,总的人员配备要求和风险分析等,但不涉及具体的细节。

按照 IEEE/ANSI 标准 829/1983,主确认测试计划包含目的和大纲两方面内容。

目的 :规定测试活动的范围、方法、资源要求和进度安排等。

大纲包含以下一些内容:

- 测试计划标识。
- 测试概述。

- 测试项目。
- 要测试的特征。
- 不要测试的特征。
- 测试方法。
- 测试任务。
- 测试环境要求。
- 进度安排。
- 项目通过/不通过的标准。
- 测试停止标准。
- 测试可交付文档。
- 风险和偶然事件。
- 人员配备和培训要求。
- 审批。

此外,对每一个确认测试活动(单元测试、集成测试、可用性测试、功能测试、系统测试和验收测试),还必须制定一个或多个详细确认测试计划。制定详细确认测试计划的目的是为了说明应该如何进行确认测试活动。上述主确认测试计划大纲中的每一项都可用于详细测试计划,只是层次更低、更详细而已。

8.5.2 测试结构设计

每个重要的软件产品都有并且只有一个测试结构规格说明,用来说明测试是如何组织的,它们是基于需求还是基于功能或是基于内部结构的测试?如何将它们进行分类?如何构建测试用例库等?测试结构设计需要考虑几方面的内容:

- 测试基础(基于需求、功能、还是内部结构)。
- 测试的分类和分类规则。
- 测试用例库的构造和命名规则等。

8.5.3 详细测试设计

详细测试设计是指定测试方法并确定测试用例的过程,包括确定测试的目标,哪些方面要优先考虑,对于相关的测试项目组,如何集中高层测试设计,但详细测试设计不给出具体的测试用例或者执行测试的步骤。实践中,从需求和功能出发,制定详细测试计划非常重要。了解代码会改变对需求的一些看法,因此制定详细测试计划时,不应过早地被软件产品的内部情况所影响。

(1) 详细测试设计考虑的问题

- 测试目标。
- 测试结构。
- 测试用例设计等。

(2) 详细测试设计的基本步骤

① 确定要测试的目标。

② 以风险为基础确定哪些项目需要优先测试。

③ 针对相关测试项目组,开发高层测试设计。

④ 根据高层测试设计,开发测试用例。

(3) 确定测试目标

确定测试目标是指确定所有要测试目标,即对需求和功能设计规格说明进行仔细研究、分解和分析,为基于功能的测试开发出测试目标清单。

对于每种类型的系统测试,考虑该类型是否适用,如果适用,则根据该类型系统测试的具体要求,利用现有的所有资源,开发基于需求的测试目标清单。

尽管测试不可能做到面面俱到,但不能忽视主要的目标,应根据风险分析挑选出应优先考虑的测试目标清单。对测试目标清单进行精炼时可采取以下几步:

① 如果基于需求和基于功能的测试目标清单是独立开发出来的,则对两份清单进行比较,删除多余的测试目标。

② 以进度安排、资源要求和不测试各项的风险为基础,对各测试目标进行低、中、高优先级筛选。评估风险时,应该意识到某种未测试的情况对于某一用户而言,可能不会造成什么损失,但对于另一用户而言则可能是灾难性的,这是由于各自不同的使用模式所造成的。

③ 对于每一测试目标清单,生成一个覆盖矩阵,用以说明测试目标和测试用例之间的关系,即哪些测试用例覆盖了哪些测试目标。

④ 对于关键软件,生成一个需求跟踪矩阵。采用需求跟踪矩阵的目的是保证对于关键软件不要漏掉什么测试目标。对每一需求都可通过对应关系找到符合该需求和功能设计要求的测试用例。表 8 – 1 给出了一个需求跟踪矩阵示范。

表 8 – 1 需求跟踪矩阵

需求	功能设计	详细设计	代码	测试用例
饭店有两个预定点	管理屏幕#2	45 页	12 485 行	34, 57, 63
服务员可从任一点预订	订购屏幕	19 页	6 215 行	12, 14, 34, 57, 92
顾客可能要求另开账单	订购屏幕	39 页	2 391 行	113, 85
顾客可从多个预定点开账单	账单打印	138 页	49 234 行	74, 104

需求跟踪矩阵将每一需求与其在功能设计中的目的,在详细设计和代码中的支持和该需求的测试用例集联系起来。如果进行了修改,则必须重新验证出处。

(4) 测试设计规格说明

对于要进行测试的实际软件产品,通常有许多测试设计规格说明。测试设计规格说明是一种概括性文档,其目的是组织和描述针对具体特征进行的测试,以帮助确定测试用例。按照 IEEE/ANSI 标准 829/1983,测试设计规格说明包括目的和大纲两部分。

目的 指出测试方法的改进之处,确定设计和相关测试所覆盖的特征,确定测试用例和测试步骤,并指定特征通过/不通过标准。

大纲:

- 测试设计规格说明标识。

- 要测试的特征。
- 采用的测试方法。
- 确定测试用例。
- 特征通过/不通过标准。

(5) 测试用例规格说明

一个测试用例可以通过多个测试设计规格说明确定,每一个测试设计规格说明又有一个或多个测试用例规格说明。

按照 IEEE/ANSI 标准 829/1983,测试用例规格说明包括目的和大纲两部分内容。

目的 :定义测试设计规格说明确定的测试用例。测试用例规格说明除了提供输入的具体值和预期的输出值,还应标明由于采用特定的测试用例而引起的对测试步骤的限制。测试用例与测试设计规格说明分开来,有利于用多种测试设计规格说明,设计测试用例。

大纲:

- 测试用例规格说明标识。
- 测试项。
- 输入要求。
- 输出要求。
- 环境要求。
- 特殊要求。
- 测试用例之间的依赖关系。

实际使用中,许多公司将这两份文档合二为一,省去了更详细的测试用例规格说明。但是,有条不紊地仔细计划测试用例,是达到测试目标的必由之路,因为:

- 组织性。即使在小型软件项目上,也可能有数千个测试用例,建立测试用例可能需要测试人员经过几个月甚至几年的时间。正确地计划和组织测试用例,有助于测试人员和其他小组成员有效地审查和使用它们。
- 重复性。项目开发期间可能会多次执行同样的测试,以保证老的软件故障得以修复,而且没有引入新的软件故障。假如没有正确的计划,就不知道最后执行的是哪个测试用例及其执行的情况如何,不便于重复原有的测试。
- 跟踪。计划执行了多少个测试用例,在软件最终版本上执行了多少个测试用例,多少个测试失败,是否有忽略的测试用例等。如果测试用例没有计划,就不能回答这些问题。
- 测试证实。在少数高风险行业中,软件测试小组必须证明确实执行了计划执行的测试。发布忽略某些测试用例的软件实际上是不合法和危险的。正确的测试用例计划和跟踪提供了一种证实测试的手段。

(6) 测试实施

测试实施是将每一个测试用例规格说明翻译成可执行测试用例的过程。测试实施可交付的文档包括:

- 测试用例。
- 测试步骤规格说明。
- 已完成的功能覆盖矩阵。

- 已完成的需求覆盖矩阵。
- 对于关键软件,已完成的需求跟踪矩阵。

(7) 测试步骤规格说明

测试步骤规格说明一步一步解释如何进行测试设置,如何开始测试,如何监视测试运行以及测试停止后如何重新开始测试。一个好的测试步骤规格说明非常重要,如何运行好一个测试用例库,不能光装在脑袋里,应该以书面形式表达出来。测试步骤规格说明包括目的和大纲两部分。

目的 确定进行测试需要的所有步骤。步骤与测试设计规格说明分开来,是因为步骤必须一步一步地执行,不应包含一些无关的细节。

大纲:

- 测试步骤规格说明标识。
- 特殊要求。
- 启动测试的步骤。
- 判断测试结果的标准,例如用秒表还是凭眼睛判断。
- 偶然事件处理步骤。
- 测试执行步骤。

上面介绍的几种测试计划规格说明之间的关系如图 8 – 1 所示。对于主测试计划,其创建过程比结果文档更重要。以下 3 个等级逐次为测试设计规格说明、测试用例规格说明和测试步骤规格说明。离主测试计划越远,侧重点就越倾向于书面文档,而不是创建过程。最低级计划变为执行测试的一步步指示,其清晰、简洁和有组织性成为关键的问题,这对测试人员每天或每小时实施测试更为实用。

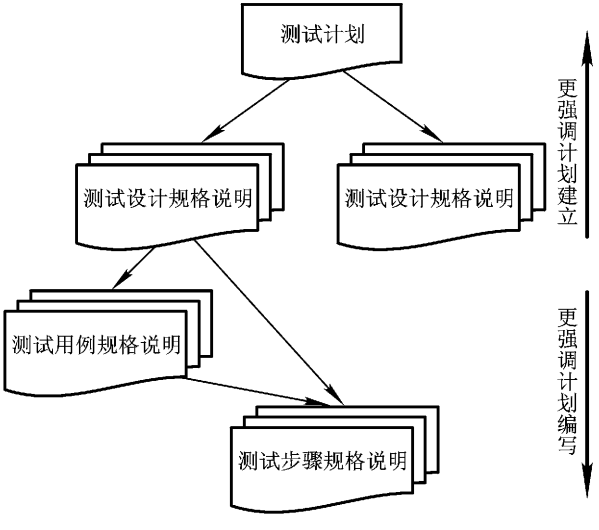


图 8 – 1 测试计划规格说明之间的关系

8.5.4 测试执行和事故报告

1. 测试执行

测试执行是执行所有或挑选的测试用例并对其结果进行观察的过程。包括：

- 测试用例选择。
- 执行前设置、执行后分析。
- 记录测试活动、结果和事件。
- 判断故障是由软件错误还是由测试本身的错误引起。
- 测量内部逻辑覆盖率。

测试执行的主要可交付文档包括测试记录、测试事故报告和逻辑覆盖报告。

2. 测试记录

测试记录保留了测试执行的有关细节，涉及测试记录什么，如何记录，记录保留多长时间，说明某个问题是一天能解决的还是需要 1 个月的时间返工等。

按照 IEEE/ANSI 标准 829—1983，测试记录包括目的和大纲两部分内容。

目的 按时间顺序记录测试执行的有关细节。

大纲：

- 测试记录标识。
- 测试描述。
- 事件记录。

3. 事故报告

(1) 报告软件故障的基本原则

事故报告是软件故障(即问题、缺陷、错误)报告的别名，其中最重要的部分是事故描述，这一部分不仅要描述出现的问题，还应将预期结果与实际结果进行比较。报告软件故障的基本原则是：

① 尽快报告软件故障。软件故障发现得越早，在进度中留下的修复时间就越多，修复软件故障的风险就越小。软件故障修复风险随时间的推移大大增加，例如，如果在软件发布之前几个月从帮助文档中找出一个令人汗颜的错别字，该软件缺陷被修复的可能性极高。图 8－2 显示了时间和故障修复之间的关系，说明软件故障发现得越晚，越不可能被修复，特别是对一般的软件故障。

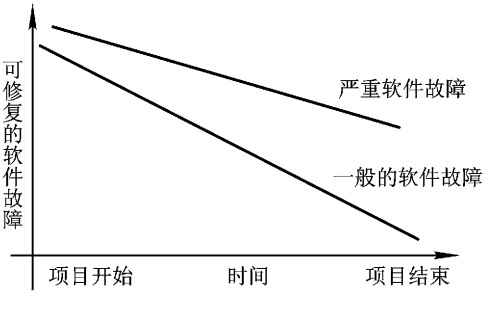


图 8－2 时间和故障修复之间的关系

② 有效描述软件故障。有效的软件故障描述包括以下几个特征：

- 短小。只描述事实和演示软件故障的必要细节。
- 单一。每一个事故报告只针对一个软件故障。
- 明显和通用。使用容易看懂、简单易行的步骤描述软件故障。
- 再现。软件故障报告必须展示其再现能力，即按照预定步骤可以使软件故障再现。

③ 报告软件故障时不做评价，只针对产品本身陈述事实。

④ 完善软件故障报告,保证软件故障被正确报告。

(2) 软件故障的严重性和优先级

不管软件故障来源于何处,任何软件故障报告都必须由提供者注明故障的严重性和优先级别。严重性表示软件故障的恶劣程度,反映其对产品和用户的影响。优先级表示修复故障的重要程度和应该何时进行修复。

下面的严重性和优先级常用划分方法有助于更好地理解两者之间的差别。有些公司使用 10 个等级,而有些公司只使用 3 个等级。然而,不论使用多少个等级,目标都是一致的。

① 严重性:

- 系统崩溃、数据毁坏、数据丢失。
- 遗漏功能、操作性错误、错误的结果。
- 小问题、用户界面布局问题、罕见故障、错别字。
- 建议。

② 优先级:

- 停止进一步测试,立即修复。
- 在产品发布之前必须修复。
- 如果时间允许,应该修复。
- 可能会修复,但也可能发布。

极少发生的数据毁坏故障应该划分为严重性 1,优先级 3。导致用户电话求助的安装指示错别字应该划分为严重性 3,优先级 2。只要一启动就导致系统崩溃的故障属于什么等级?可能是严重性 1,优先级 1。如果认为某按钮应该向页面下方再移动一点,那么它应该属于严重性 4,优先级 4。

在确定优先服务,判断产品发布前是否准备就绪,定义各种质量度量以及决定哪些软件故障应该及时进行修复,以何种顺序进行修复等方面,故障严重性和优先级概念都是非常重要的。如果要修复 25 个软件缺陷,应该从最严重的故障开始,而不只是修复那些最容易的。同样,如果两个项目管理员——一个管理游戏软件的开发,另一个管理心脏监视仪软件的开发,虽然他们可能使用同样的故障信息,但是根据同样的故障信息却可能会做出不同的决定:一个可能会选择使软件更美观,执行速度更快的做法;而另一个可能会选择使软件尽量可靠的做法。严重性和优先级信息是他们用于做这些决定的依据。

从开发者来看,一个故障的严重性有两方面的含义:出现该故障的概率和该故障真正出现时产生的影响。从用户角度来看,用户通常只关心后一种情况。

(3) 软件故障报告

按照 IEEE/ANSI 标准 829/1983,软件故障报告包括目的和大纲两部分内容。

目的:记录需要进一步调查的测试执行事件。

大纲:

- 软件故障报告标识。
- 概述。
- 软件故障描述。
- 软件故障影响。

8.6 测试评估

(1) 测试评估的内容

测试评估包括以下几个方面：

① 测试覆盖评估。测试覆盖评估是对测试用例集合进行的全面性评价，并决定是否需要补充测试用例。全面性评价基于当前各个层次的测试覆盖性，即功能覆盖(采用功能覆盖矩阵)、需求覆盖(采用需求覆盖矩阵)和逻辑覆盖(采用覆盖测量结果)。这项工作的具体可交付文档是合适的补充测试用例。

② 软件故障评估。软件故障评估则根据测试的执行对软件质量进行评价，并决定是否需要开发补充测试用例。通常认为，20% 的代码最可能发生故障，越快将这 20% 定位越好，这样可以预测剩余的故障数。软件故障评价基于发现故障的数量，其性质和严重程度，出现故障的区域以及故障检测率等。这项工作的具体可交付文档是需要补充的测试用例。

③ 测试有效性评估。测试有效性评估是对照测试停止标准，对当前测试工作的整体有效性进行评价，以及决定是停止测试还是增加测试用例并继续测试的过程。有效性评价基于测试覆盖评估和软件故障评估。主要考虑的问题有：

- 决定停止测试还是继续测试。
- 如果决定继续测试，则需要补充哪些测试。
- 如果决定停止测试，如何撰写测试总结报告。

这里可以将测试用例分成以下 4 类：

- 计划的：计划开发的测试用例。
- 可用的：可用于执行的测试用例(可用的测试用例少于或等于计划的测试用例)。
- 执行的：已执行的测试用例(已执行的测试用例少于或等于可用的测试用例)。
- 通过的：执行过程中没有发现故障的测试用例(通过的测试用例少于或等于已执行的测试用例)。

测试过程中，分别计算这 4 类测试用例的数量，并将这 4 组数字对应于时间(星期、天等)绘制成曲线，如图 8－3 所示。通过对这些曲线的分析可以获得实用的测试管理信息，也可以按时

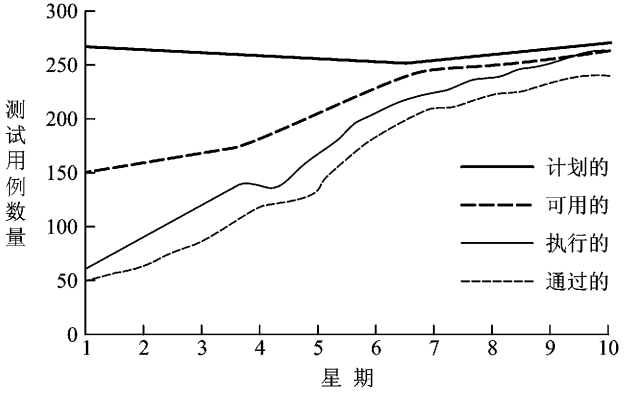


图 8－3 测试执行

间跟踪或根据通过的测试用例与计划的测试用例之比进行跟踪。例如,可以决定只有当通过的测试用例与计划的测试用例之比超过 97% 时才允许停止测试。

(2) 测试总结报告

最终文档是测试总结报告,对确认测试活动的结果作出概述,包括 :计划内容、从覆盖角度已完成的工作量、发现的故障数及其严重程度等。

按照 IEEE/ANSI 标准 829/1983,测试总结报告包括目的和大纲两部分内容。

目的 :总结与一个或多个测试设计规格说明相关的测试活动结果并提供基于这些结果的评估。

大纲 :

- 测试总结报告标识。
- 测试概要。
- 测试活动概述。
- 测试综合评价。
- 测试结果概述。
- 测试评估。
- 测试审批。

8.7 用户手册

用户手册是事关软件产品成败的主要因素之一,其重要性决不低于代码本身。对于用户来说,如果用户手册上说明要做什么事情,但用户照着去做,却不能成功,那么即使代码正确也无济于事。因此,必须在开发过程中检查文档草稿以确保文档的正确性、可理解性和完整性,必须将用户手册当做产品的重要部分来对待。必须采用验证测试(包括计划和报告)的概念和方法对手册进行综合测试,其主要目标是找出功能设计规格说明和用户手册之间的差异。手册中的范例都应该做为手册测试的一部分接受测试,以判断它们运行起来是否与描述的一样。

8.8 IEEE/ANSI 测试文档概述

(1) 测试计划和规格说明的文档结构

下面给出用于测试计划和规格说明的所有文档之间的相互关系如图 8 -4 所示。

- SQAP 软件质量保证计划,每个软件测试产品有一个 SQAP。
- SVVP 软件验证和确认测试计划,每个 SQAP 有一个 SVVP。
- VTP 验证测试计划,每个验证活动有一个 VTP。
- MTP 主确认测试计划,每个 SVVP 有一个 MTP。
- DTP 详细确认测试计划,每个活动有一个或多个 DTP。
- TDS 测试设计规格说明,每个 DTP 有一个或多个 TDS。
- TPS 测试步骤规格说明,每个 TDS 有一个或多个 TPS。
- TCS 测试用例规格说明,每个 TDS/TPS 有一个或多个 TCS。

- TC **测试用例,每个** TCS **有一个** TC。

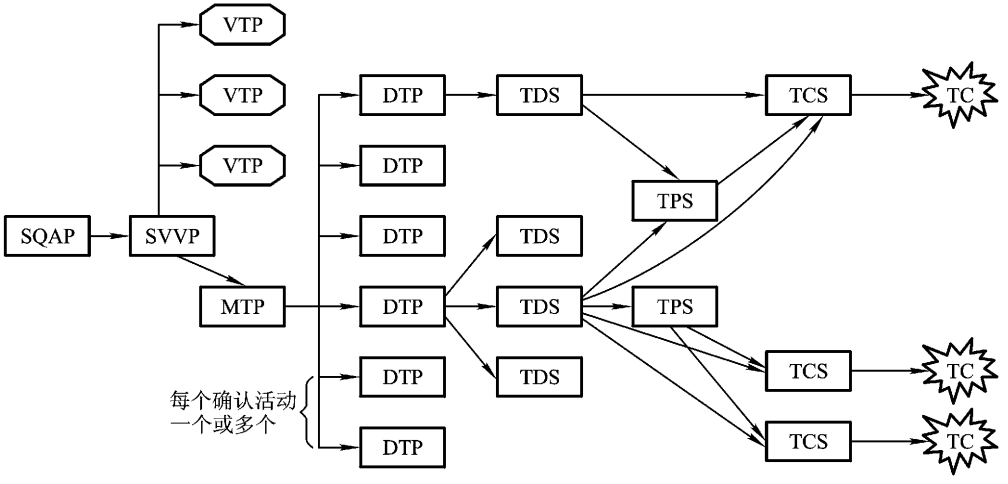


图 8 – 4 **测试计划和规格说明的文档结构**

由图 8 – 4 **可以看出 :每个软件产品都有一个软件质量保证计划,每个软件质量保证计划有一个软件验证和确认测试计划,每个软件验证和确认计划有一个主确认测试计划。每个验证测试活动有一个验证测试计划,每个确认测试活动有一个或多个详细确认测试计划,每个详细确认测试计划有一个或多个测试设计规格说明,每个测试设计规格说明有一个或多个测试步骤规格说明,每个测试设计规格说明/测试步骤规格说明有一个或多个测试用例规格说明,每个测试用例规格说明有一个测试用例。**

(2) 测试报告的文档结构

图 8 – 5 **给出了测试报告的文档结构。**

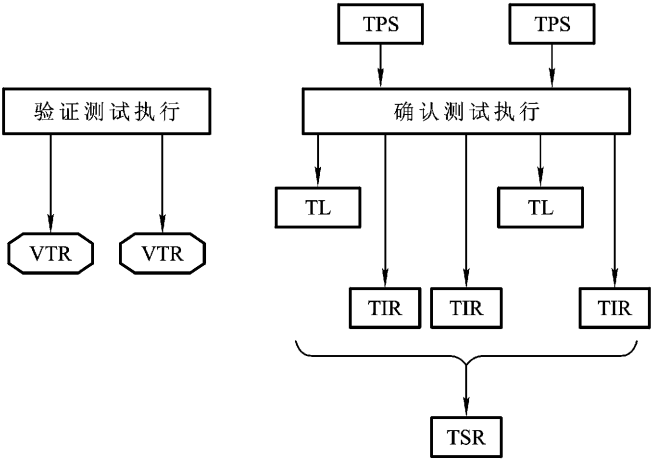


图 8 – 5 **测试报告的文档结构**

VTR 验证测试报告,每个验证活动一个。

TPS 测试步骤规格说明。

TL 测试记录,每个测试执行一份。

TIR 测试事故报告。每个事故一个。

TSR 测试总结报告。

8.9 软件生存周期各阶段的测试任务与可交付的文档

软件生存周期按瀑布模型可以分为:需求、功能设计、详细设计、编码、测试和运行维护 6 个阶段,不同阶段可能在一定程度上有重复,但各个阶段的结束必须按一定顺序进行,比如提交可交付文档、审批、签字等。

8.9.1 需求阶段

(1) 测试输入

- 软件质量保证计划(任选)。
- 需求(来自开发)。

(2) 测试任务

- 制定验证和确认测试计划。
- 对需求进行分析。
- 对需求进行审核。
- 分析并设计基于需求的测试,构造相应的需求覆盖或跟踪矩阵。

(3) 可交付的文档

- 软件验证测试计划。
- 验证测试计划(针对需求)。
- 验证测试报告(针对需求)。

8.9.2 功能设计阶段

(1) 测试输入

功能设计规格说明(来自开发)。

(2) 测试任务

- 功能设计验证和确认测试计划。
- 分析功能设计规格说明。
- 审核功能设计规格说明。
- 可用性测试设计。
- 分析并设计基于功能的测试,构造相应的功能覆盖矩阵。
- 实施基于需求和基于功能的测试。

(3) 可交付的文档

- (主确认)测试计划。

- 验证测试计划(针对功能设计)。
- 验证测试报告(针对功能设计)。

8.9.3 详细设计阶段

(1) 测试输入

详细设计规格说明(来自开发)。

(2) 测试任务

- 详细设计验证测试计划。
- 分析详细设计规格说明。
- 审核详细设计规格说明。
- 分析并设计基于内部的测试。

(3) 可交付的文档

- 详细确认测试计划。
- 验证测试计划(针对详细设计)。
- 验证测试报告(针对详细设计)。
- 测试设计规格说明。

8.9.4 编码阶段

(1) 测试输入

代码(来自开发)。

(2) 测试任务

- 代码验证测试计划。
- 分析代码。
- 验证代码。
- 设计基于外部的测试。
- 设计基于内部的测试。

(3) 可交付的文档

- 测试用例规格说明。
- 需求覆盖或跟踪矩阵。
- 功能覆盖矩阵。
- 测试步骤规格说明。
- 验证测试计划(针对代码)。
- 验证测试报告(针对代码)。

8.9.5 测试阶段

(1) 测试输入

- 要测试的软件。
- 用户手册。

(2) 测试任务

- 制定测试计划。
- 审查由开发部门进行的单元和集成测试。
- 进行功能测试。
- 进行系统测试。
- 审查用户手册。

(3) 可交付的文档

- 测试记录。
- 测试事故报告。
- 测试总结报告。

8.9.6 运行/维护阶段

(1) 测试输入

- 已确认的问题报告。
- 软件生存周期。软件生存周期是一个重复过程。如果对软件进行了修改,开发和测试活动都应回到与修改相对应的生存周期阶段。例如,如果软件增加了新功能,则要求制定新的功能设计规格说明,而测试过程也必须回到功能设计阶段并由此按顺序继续进行下去。

(2) 测试任务

- 监视验收测试。
- 为确认的问题开发新的测试用例。
- 对测试的有效性进行评估。

(3) 可交付的文档

升级的测试用例库。

小结

成功的测试需要有一定的方法,包括条例、结构、分析和度量。没有组织、目标、随便的测试注定要失败。软件测试计划是软件测试小组与产品开发小组交流意图的主要方式。制定测试计划的目标是:规定测试活动的范围、方法、资源和进度,明确正在测试的项目,要执行的主要测试任务,每个任务的责任人以及与计划相关的风险。

软件测试文档对于提高软件质量,保证软件正常运行有着十分重要意义。《计算机软件测试文档编制规范》国家标准给出了具体的软件测试文档编制建议,其中包括:测试计划、测试设计规格说明、测试用例规格说明、测试步骤规格说明、测试事件报告和测试总结报告等方面的建议。

第 8 章习题

1. 制定测试计划的目的是什么,给出制定测试计划时应考虑的一些常用测试资源。

2. 软件故障可能不修复的几个原因是什么？

3. 哪些基本原则可以用于软件故障报告，从而使软件故障获得较大的修复机会？

4. 假设测试 Windows 的计算器程序，发现 $1+1=2$，$2+2=5$，$3+3=6$，$4+4=9$，$5+5=10$，$6+6=13$，写出一个有效描述该问题的软件故障报告。

5. 测试评估包括几方面的内容，评估的主要依据是什么？

第 9 章　面向对象的软件测试

面向对象是一种全新的软件开发技术，正逐渐取代传统的面向过程的软件开发方法，被看成是解决软件危机的新兴技术。尽管面向对象的分析、设计方法有助于构造良好的系统结构，其程序设计语言有利于提高软件的可重用性、可扩充性、互操作性及可行性，但如何提高软件质量仍然是软件工程学的一个主要问题。

使用面向对象的软件开发技术，代码重用率较高，但要求更严格的测试，以避免故障的繁衍。面向对象软件测试的主要目标，仍然是用尽可能低的测试成本和尽可能少的测试用例，尽可能多地发现软件中隐藏的各种故障。但是，面向对象程序设计语言中特有的封装、继承和多态等机制，给面向对象的测试带来了新的问题，增加了测试的难度。

本章重点：

- 面向对象的概念
- 面向对象的软件测试与传统软件测试之间的不同
- 面向对象的单元测试
- 面向对象的集成测试

9.1　面向对象的概念

面向对象的程序设计以对象、消息、接口、类、继承、动态绑定等基本概念为核心。关于这些概念，大家都比较熟悉，这里主要从测试的角度来考虑。例如，操作和方法（或者成员函数）对大多数程序员来说并没有太大的区别，然而对于测试人员来说，区别是显然的，因为测试一个操作的步骤在某种程度上与测试一个方法的步骤不同，前者是类声明的一部分同时也是操纵对象的一种手段，后者则是实现某个操作的一些代码。

9.1.1　对象

对象是封装了描述其属性的数据以及可以施加在其上的操作的封装体。属性表示对象的状态，操作表示对象的行为，即对象是一个可以操作的实体，它既包含了特定的数据，又包含了操作这些数据的代码。

当一个程序被执行时，对象被创建、修改、访问或删除。程序运行时，对象的行为是否符合规格说明，对象与其他相关的对象能否协同工作，都是面向对象软件测试所关注的焦点。

从测试的角度看，对象具有以下几个属性：

- 对象隐藏了信息。这一点使得对象信息的改变有时很难被观察，也使得测试结果的检查难度加大。
- 对象的生命周期。对象具有生命周期。在对象生命周期的不同阶段，为了确定对象的状态是否符合它的生命周期，对象可能会被从各个不同的方面进行检测。过早地创建一个对象或

过早地删除一个对象,都是造成软件故障的原因。

- 对象的状态。在对象的生存周期中,对象都有一个状态,对象的状态是多变的,这也可能是造成软件故障的根源。

9.1.2 消息

消息描述了对象执行操作的规格说明。对象之间通过发送消息启动相应的操作,通过修改对象的状态,实现系统状态间的相互转换。面向对象的程序通过一系列对象协同工作来解决问题,这一协作则是通过对象之间相互传送消息来实现的。

从测试的角度看,消息具有以下几个属性:

- 消息发送者。发送者可以决定何时发送消息,而且可能会做出错误的决定。
- 消息接收者。接收者可能接收到非预期的消息,当它接收到一条非预期的消息时,接收者可能会对此做出不正确的反应。
- 消息可能包含有实际参数。在处理一条消息时,参数可能被接收者使用或改变。如果传送的参数是对象,那么在消息被处理前后,对象必须处于正确的状态,必须实现接收者所期望的接口。

9.1.3 接口

接口是行为声明的集合。行为被集中在一起,并通过单个的概念定义一些相关的动作。比如,一个接口可能描述了关于对象如何移动的一系列动作。

从测试的角度看,接口具有以下几个属性:

- 接口封装了操作的说明。如果接口包含的行为和类的行为不相符,那么对这一接口的说明就不能令人满意。
- 接口不是孤立的,它与其他的接口和类有一定的关系。

9.1.4 类

类是具有相同特征的对象集合。对象是类的实例。面向对象程序执行的基本元素是对象,类是用来定义对象的。

- 在类声明中,定义了类的每个对象能做什么。
- 在类实现中,定义了类的每个对象如何做它们能做的东西。

类测试是面向对象测试过程中最重要的一个测试,在类测试过程中要保证测试那些具有代表性的类的成员函数。

从测试的角度来研究类,可以得出面向对象设计和实现中一些潜在的故障原因:

- 类的说明包含用来构造实例的一些操作,这些操作可能会导致不正确地初始化新的实例。
- 类在定义自己的行为和属性时,也依赖于其他与之共同协作的类。如果在定义类时使用了包含有不正确实现的其他类,就会使类发生故障。
- 类的实现可能不支持所有要求的操作,或者执行了一些错误的操作。

9.1.5 继承

继承是类之间的一种联系,它允许新类在已有类的基础上定义。一个类对另一个类的依赖,使得已有类的说明和实现可以被重复使用。

从测试的角度看,继承具有以下几个属性:

- 继承提供了一种机制,通过这种机制,潜在的故障可能从一个类传递到它的派生类中。测试类的时候应尽早消除这种故障,以免这些故障被传递到其他类中。
- 继承还提供了另一种机制,这种机制能使人们重复使用相同的测试方法。因为子类是从它的父类继承过来的,所以子类也就继承了父类的说明和实现。因此也就可以用测试父类的方法对子类进行测试。

9.1.6 动态绑定

动态绑定是类具有不同表现形式的一种现象,动态绑定使得设计和编码比以前更加抽象。

从测试的角度看,动态绑定具有以下几个属性:

- 动态绑定允许系统通过增加类来进行扩展,而不是修改已存在的类。当然在扩展中也会出现出乎预料的交互关系。
- 动态绑定允许任何操作都可以包括一个或多个类型不确定的参数,这样就增加了应该测试的实际参数的种类。

9.2 面向对象的测试与传统软件测试的区别

面向对象软件特有的继承、封装和动态绑定以及和传统软件之间的差异给面向对象的软件测试提出了一系列新的问题。例如,在传统的面向过程的程序中,对于函数

y = Function(x);

只需要考虑函数 Function() 的行为特征,而在面向对象的程序中,就不得不同时考虑基类函数 Base::Function() 和继承类函数 Derived::Function() 的行为特征。

面向对象的程序结构已不再是传统的功能模块结构。封装是面向对象软件的一个重要特征,把数据及对数据的操作封装在一起,限制了对象属性对外的透明性和外界对它的操作权限,在一定程度上避免了传统软件中对数据的非法操作,有效地防止了故障的扩散。但是,封装机制也给测试数据的生成、测试路径的选取以及测试结果的分析带来了困难。

继承与动态绑定机制是面向对象实现的主要手段。继承允许子类不仅可以共享父类中定义的数据和操作,还可以定义新的特征,然而,子类是在一个新的环境中,父类的正确性不能保证子类的正确性。此外,继承可以使代码的重用率提高,同时也使故障的传播概率增加。所以,研究继承关系的测试方法及策略是面向对象的测试研究的一个重点和难点。

面向对象的动态绑定明显增加了系统运行中可能的执行路径,而且给面向对象软件带来了严重的不确定性,这种不确定性和骤然增加的路径组合无疑为测试覆盖率的满足带来了新的挑战。

面向对象软件的依赖性问题,是由于面向对象软件中存在诸如继承、关联、类嵌套和动态绑

定等关系引起的。一个类不可避免地要依赖于其他类。

(1) 传统软件中存在的依赖关系有：

- 变量间的数据依赖。
- 模块间的调用依赖。
- 变量与其类型间的定义依赖。
- 模块与其变量间的功能依赖。

(2) 面向对象软件除了存在上述依赖关系外,还存在以下的依赖关系：

- 类与类间的依赖。
- 类与操作间的依赖。
- 类与消息间的依赖。
- 类与变量间的依赖。
- 操作与变量间的依赖。
- 操作与消息间的依赖。
- 操作与操作间的依赖。

例如,Interview 类库中,有 122 个类、400 多个继承、聚合、关联等关系,并有 100 多个操作,这 122 个彼此相关的类构成了一个强连通循环图。因此,测试人员在制定测试计划,设计测试用例时,需要充分考虑上述各种可能的依赖关系,这大大增加了面向对象软件测试的复杂度。

9.3 面向对象软件测试

面向对象的开发模型将软件开发分为面向对象分析,面向对象设计和面向对象编程 3 个阶段。

分析是软件开发活动的一个阶段。传统的面向过程的分析是一个功能分解的过程,是把一个系统看成是可以分解的功能集合,这种传统的功能分解分析法的着眼点在于一个系统需要什么样的处理方法和过程。而面向对象分析则从系统能完成的功能,以及对象间的相互关联关系为核心,分析主要围绕对象的分类,各个相关属性和操作的标识,类和实例之间的关系,各种对象的行为特征等方面进行。

设计描述软件如何才能满足需求。结构化的设计方法,通常是一种面向过程的设计方法。面向对象的设计是面向对象分析的进一步细化和扩充,重点在于说明项目的实施方案确定类和类结构。设计不仅要满足当前需求分析的要求,更重要的是要能方便地实现功能的重用和扩充,以不断适应用户的要求。在此基础上,进一步归纳出适用于面向对象编程语言的类和类结构,最后生成源程序代码。实际上,源程序代码是面向对象设计模型用编程语言进行描述的一种准确解释。

使用面向对象开发模型的一个优点是:分析模型可以映射成设计模型,而设计模型又可以映射成源程序代码。因此,可以在分析阶段开始测试,将分析阶段的测试提炼以后可用于设计阶段的测试,设计阶段的测试经过提炼以后又可用于实现阶段的测试。这意味着测试过程可以与开发过程交替进行。对分析和设计模型进行测试的好处还有：

- 测试用例可以在过程中更早地确定,甚至在需求确定以前。尽早测试可以帮助分析者和

设计者更好地理解和表达需求,并确保特定的需求是‘可测试的”。

- 缺陷可以在开发过程的早期检测出来,从而节省了时间、金钱和精力。大家都知道,问题检测出来得越早,解决问题就越简单,开销就越少。
- 可以在开发的早期检查测试用例的正确性。测试用例的正确性是非常关键的,如果能在项目中尽早确定测试用例并运用到模型上,那么测试人员对需求的任何误解都能被尽早地纠正。换句话说,这有助于确保测试人员和开发人员对系统需求的理解保持一致。

面向对象程序的测试主要针对编程风格和源程序代码进行测试,测试内容主要体现在面向对象单元测试和面向对象集成测试中。面向对象单元测试针对程序内部具体单一的功能模块进行测试。面向对象集成测试则针对系统内部的相互服务进行测试,如成员函数间的相互作用,类间的消息传递等。面向对象系统测试则主要以用户需求为测试标准。

9.4 类测试

在测试类的功能实现时,应该首先保证类成员函数的正确性。单独看待类的成员函数,与面向过程程序中的函数或过程没有本质的区别,因此可以将类的成员函数看做单元,将面向对象单元测试归结为传统过程的单元测试,那么几乎所有传统的单元测试中所使用的方法,都可在面向对象的单元测试中使用,如等价类划分、因果图、边界值分析、逻辑覆盖等。传统过程的单元测试需要桩或驱动模块,类似地,面向对象的单元测试,也必须提供能够实例化的桩类,以及起驱动作用的‘主程序”类。

面向对象编程的固有特性使得对成员函数的测试,又不完全等同于传统的函数或过程测试,尤其是继承和多态特性,使子类继承父类的成员函数时,出现了传统测试中没有遇到过的问题,因此,还需要考虑以下几个问题。

(1) 继承的成员函数是否都不需要测试

对父类中已经测试过的成员函数,有两种情况需要在子类中重新进行测试:

- 继承的成员函数在子类中做了改动。
- 成员函数调用了改动过的成员函数。

例如,假设父类 Base 有两个成员函数 ::Inherited() 和 Redefined(),子类 Derived 只对 Redefined() 做了改动。显然,Derived::Redefined() 需要重新测试。对于 Derived::Inherited(),如果它调用了 Redefined(),就需要重新测试,反之,不需要。

(2) 对父类的测试能否照搬到子类

基于上面的假设,Base::Redefined() 和 Derived::Redefined() 已经是不同的成员函数,它们有不同的操作和说明。因此,照理应该对 Derived::Redefined() 重新进行测试分析,设计测试用例,但由于面向对象的继承使得两个函数很相似,故只需在 Base::Redefined() 的测试要求和测试用例集上添加上对 Derived::Redfined() 新的测试要求和增补一些测试用例即可。

在对类进行单元测试时,类中特定的实现细节很容易变化,一旦被测试类的某些实现细节发生了变化,原先的测试就要进行相应的更改,否则就失去了意义,这种频繁更改的代价是巨大的。类是一种抽象,它反映了更高层面的内聚性,应该有明确的职责和定义良好的接口,它们相对于内部实现细节来说要稳定的多,并且我们要验证的正是这个类是否具备了它的职责。因此,在对

类进行测试时,主要针对类接口来验证类是否实现了它的职责。

(3) NextDate 函数的面向对象实现

图9-1和图9-2给出了前面 NextDate 函数的一种面向对象实现中的类和类的对象实例。每一个类都只封装了自己所需要的信息,其他类的信息则通过消息获得。testIt 类实例化为一个测试日期,然后对其加1,并打印出结果日期。Date 对象由 Month、Day 和 Year 实例组成。Date.increment 成员函数向 Day 对象发送消息,查看其是否能够增1(包括月末问题),向 Month 对象发送消息,查看其能否增1(包括年末问题)。

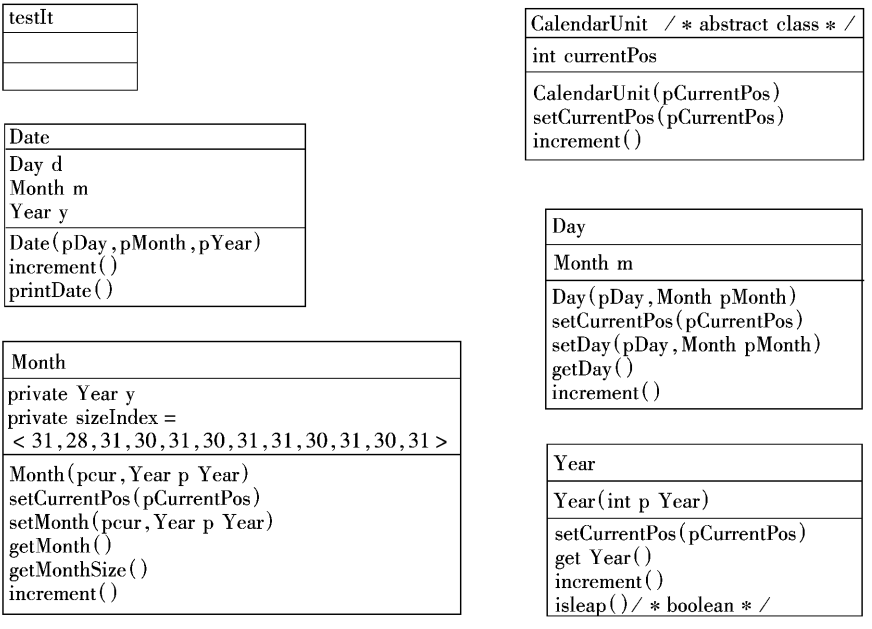


图9-1 O-Ocalender 中的类

图9-2 类继承与聚合

下面给出每个类的类职责及伪代码。

① CalendarUnit 类

职责 提供一个操作在所继承的类中设置取值,提供一个布尔操作说明所继承类中的属性是否可以增1。

```
/*   Collaborates with :Day,Month,and Year  */
Class CalendarUnit      /* abstract class  */
 {
    int currentPos;
    Calendar Unit (pCurrentPos)
     {
```

```
            CurrentPos = pCurrentPos
        }
        setCurrentPos(pCurrentPos)
        {
            currentPos = pCurrentPos
        }
        Abstract protected boolean increment()
    }
```

② testIt 类

职责:用做测试驱动器,即创建一个测试日期对象,然后请求该对象对其本身加 1,最后打印新值。

```
    /* Collaborates with:Date */
    Class testIt
    {
        main()
        {
            testdata = instantiate Date(testMonth,testDay,testYear)
            testdata.increment()
            testdate.printDate()
        }
    }
```

③ Date 类

职责:Date 对象由 Day、Month 和 Year 对象组成。Date 对象使用所继承的 Day 和 Month 对象中的布尔增量方法对其本身加 1。如果 Day 和 Month 对象本身不能加 1(例如月份或年的最后一天),则 Date 根据需要重新设置 Day 和 Month。如果是 12 月 31 日,则 Year 也要增 1。printDate 操作使用 Day,Month 和 Year 对象中的 get()成员函数,并以 mm/dd/yyyy 格式打印出日期。

```
    /* Collaborateswith:testIt,Day,Month,and Year */
    class Date
    {
        private Day d
        private Month m
        private Year y
        Date(pMonth,pDay,pYear)
        {
            y = instantiate Year(pYear)
            m = instatiate Month(pMonth,y)
            d = instandate Day(pDay,m)
        }
```

```
    increment()
     {
        if (NOT(d. increment()))
            if (NOT(m. increment()))
             {
               y. increment()
               m. setMonth(1,y)
             }
            else   d. SetDay(1,m)
    }
    printDate()
     {
      Output(m. getMonth() +"/"+ d. getDay() + "/"+ y. getYear())
    }
}
```

④ Day 类

职责：Day 对象有一个私有 Month 属性，用来查看 Day 取值是要加 1 还是复位。Day 对象也提供 get()和 set()成员函数。

```
/* Collaborates with:Month */
class Day::CalendarUuit
 {
    private Month m
    Day(pDay,Month pMonth)
     {
        setDay(pDay,pMonth)
    }
    setDay(pDay, Month pMonth)
     {
        setCurrentPos(pDay)
        m = pMonth
    }
    getDay()
     {
    return currentPos
    }
    Boolean increment()
     {
        currentPos = currentPos + l
```

```
            if (currentPos < = m. getMonthSize())
                return true
            else return false
        }
    }
```

⑤ Month 类

职责：Month 对象有一个值属性，用做月份最后一天的值的数组下标（例如 1 月的最后一天是 31，2 月的最后一天是 28 等）。Month 对象提供 get()和 set()服务，以及所继承的布尔增量方法。2 月 29 日的判决，由发给 Year 对象的 isLeap 消息决定。

```
/* Collaborates with :Year */
class Month :: CalendarUnit
 {
    Private Year y
    private sizeIndex = <31,28,31,30,31,30,31,31,30,31,30,31>
    Month(pcur,Year pYear)
     {
        setMonth(pCurrentPos,Year pYear)
    }
    setMonth(pcur,Year pYear)
     {
        setCurrentPos(pcur)
          y = pYear
    }
    getMonthsize()
     {
        return currentPos
    }
    getMonthSize()
     {
        if (y. isleap())
            sizeIndex[1] =29
        else      sizeIndex[1] =28
        return sizeIndex[CurrentPos - 1]
    }
    Boolean increment()
     {
        currentPos = currentPos + 1
        if (CurrentPos > 12)
```

```
            return false
          Else return true
      }
  }
```

⑥ Year 类

职责：除了一般的成员函数 get()和 set()外，如果测试日期是任意年的 12 月 31 日，Year 对象自己也要增 1。Year 对象提供了一个布尔服务，说明当前值是否对应闰年。

```
/* Collaborates with:(no external messages sent) */
    class Year::CalendarUnit
     {
      Year(pYear)
       {
        setCurrentPos(pYear)
      }
      getYear()
       {
          return      currentPos
      }
      boolean   increment()
       {
          currentPos = currentPos + 1
          return true
      }
      Boolean isleap();
       {
        if(((currentPos MOD 4 =0) AND NOT(currentPos MOD 400 =0)) OR(currentPos MOD 400 =0))
          return true
        else return false
      }
}
```

⑦ 成员函数 Date. increment 的单元测试

Date. increment 对 Day 的处理可以划分为 3 个等价类：

D1 = {Day：1 < = Day < 月的最后一天}

D2 = {Day：Day 是非 12 月的最后一天}

D3 = {Day：Day 是 12 月的最后一天}

这些等价类定义比较松散，尤其是 D1 类，没有说明是哪个月份。因为面向对象软件的封装，这些问题可以被忽略，实际上问题被转化为 Month. increment 操作的测试问题。

类函数成员的正确行为只是类能够实现其要求功能的基础，类成员函数间的作用和类之间的相互调用是单元测试无法确定的。因此，需要进行面向对象的集成测试。

9.5 面向对象的集成测试

传统的集成测试是通过自底向上或自顶向下集成完成功能模块的集成测试,一般可以在部分程序编译完成以后进行。但对于面向对象程序,相互调用的功能是分布在程序的不同类中,类通过消息相互作用申请并提供服务。类相互依赖极其紧密,根本无法在编译不完全的程序上对类进行测试。所以,面向对象的集成测试通常需要在整个程序完成编译以后进行。此外,面向对象的集成测试需要进行两级集成:一是将成员函数集成到完整类中,二是将类与其他类集成。

面向对象的集成测试能够检测出单元测试无法检测出的那些类相互作用时才会产生的故障。单元测试可以保证成员函数行为的正确性,集成测试则只关注于系统的结构和内部的相互作用。

面向对象的集成测试可以分成静态测试和动态测试两步进行。

静态测试主要针对程序结构进行,检测程序结构是否符合设计要求,通过静态测试方式处理由动态绑定引入的复杂性。动态测试则测试与每个动态语境有关的消息。面向对象集成测试的动态视图更加重要。

以类为单元进行测试,也是一种常用的单元测试方法。下面主要讨论基于类的面向对象的集成测试。

在采用统一建模语言定义的面向对象软件中,协同图和序列图是集成测试的基础。协同图描述类之间(部分)的信息传递关系,协同图则既支持成对出现的类集成测试,也支持相邻类的集成测试。

成对集成是对发送消息或接收消息的独立“相邻”类进行的测试。就类向其他类发送/接收消息而言,其他类必须表示为桩。所有这些额外的工作都使得成对的类集成难以接受,这与过程单元的成对集成一样。根据图 9-3 所示的协同图,可以得到以下要集成的类对:

testIt 和 Date 需要为 Year、Month 和 Day 类建立桩。

Date 和 Year 需要为 testIt、Month 和 Day 类建立桩。

Date 和 Month 需要为 testIt、Year 和 Day 类建立桩。

Date 和 Day 需要为 testIt、Month 和 Year 类建立桩。

Year 和 Month 需要为 Date 和 Day 类建立桩。

Month 和 Day 需要为 Date 和 Year 类建立桩。

以协同图为基础进行面向对象集成测试的缺点是,在类级中行为模型是“状态图”。对于类级行为,“状态图”是设计测试用例的很好的途径,特别适合于以类为单元的测试。但是,一般说来“状态图”很难合并,不便在较高层次上观察行为。

图 9-3 O-Ocalender 的协同图

相邻类的集成会降低设计桩类的工作量,但是要付出降低诊断精度的代价。如果测试用例失败,则必须在更多的类中寻找故障。

图9-4 printDate 的序列图

序列图跟踪协同图的执行时间路径。图9-4给出的是打印新日期2002年1月15日部分的程序序列图。粗竖线表示类或类的实例,箭头表示类的实例按时间顺序发送的消息。伪代码实现如下:

```
1. TestDriver
      {
2. m. setMonth(1)
3. d. setDay(15)
4. y. setYear(2002)
5. Output("expected value is","1/15/2002")
6. Output("actual output is")
7. testIt. printDate()
8. }
```

语句2、3和4使用已通过单元测试的成员函数设置要向其发送消息的类的预期输出。

系统测试是独立于系统实现的,系统测试人员不需要知道实现采用的是面向过程的代码还是面向对象的代码。

小结

面向对象软件测试的主要目标,仍然是用尽可能低的测试成本和尽可能少的测试用例,尽可能多地发现软件中隐藏的故障。但是,面向对象程序设计语言中特有的封装、继承和多态等机制,给面向对象的测试带来了新的问题,增加了测试的难度。

面向对象的开发模型突破了传统的瀑布模型,将开发分为面向对象分析,面向对象设计和面向对象编程3个阶段。分析模型可以映射成设计模型,而设计模型又可以映射成程序源代码,这意味着测试过程可以与开发过程交替进行。

面向对象的单元测试可以以类为单位进行也可以以类的成员函数为单位进行。面向对象的集成测试则主要针对成员函数间的相互作用,类间的消息传递进行。面向对象系统测试是基于面向对象集成测试的最后阶段的测试,以用户需求为主进行测试。

第9章习题

1. 面向对象的开发模型与传统软件的开发模型有何不同?
2. 简述面向对象的测试与传统软件测试的区别。
3. 简述面向对象的单元测试与集成测试。

第 10 章　软件测试自动化和测试工具

大多数软件发布之前都要经过多次重复的代码测试。测试某项特性,不仅要检查前面测试中发现的软件故障是否得到真正的修复,还要检查修复过程是否引入了新的软件故障等,这意味着需要不止一次地执行测试。如果一个小型软件项目有数千个测试用例要执行,那么在时间有限的情况下,手工进行多次测试执行是不可能的。

软件测试自动化可以省去许多繁杂的工作,节省软件测试时间,提供比手工测试更好、更快的测试执行方式。因此,使用测试自动化和测试工具会对整个软件开发工作的质量、成本和周期带来非常显著的效果。

本章重点:

- 软件测试自动化
- 测试工具

10.1　测试与测试自动化

软件测试是一门技术。对于任何系统而言,都存在着大量可能的输入,测试只能执行其中很少的一部分输入,并希望通过这些有限的测试输入发现软件中存在的大多数故障。因此软件测试技术不仅要保证设计的测试用例能够发现尽可能多的软件故障,而且还要保证测试用例的设计是经济有效的。

测试自动化也是一门技术,但与测试技术存在很大区别。测试自动化希望通过自动化测试工具或其他手段,按照测试工程师的预定计划自动地进行测试,目的是减轻手工测试的劳动量,从而达到提高软件质量的目的。测试自动化的目的在于发现老的软件故障,而手工测试的目的在于发现新的软件故障。

自动化测试通常要比手工测试经济得多,其开销只是手工测试的一小部分。自动化测试的方法越好,长期使用获得的收益就越大。

要实现高效的自动测试,必须源于好的测试软件,这些测试软件是由经验丰富的测试人员精心设计的,在此基础上再应用自动化技术实现自动测试,这样可以获得建立及维护的合理开销。

软件测试可以是高质量或劣质的,这取决于测试者实现测试的技术。同样,自动化测试的质量也可以是高质量或劣质的,这取决于测试自动化者的自动化技术,包括确定如何方便地增加新的自动测试,如何维护自动测试以及如何提高测试自动化的效益等。

10.2　测试工具

随着软件测试的重要性逐步显现,选择和使用测试工具的好处尽人皆知。目前有各种各样的计算机辅助软件测试工具,可用于测试过程的许多方面。但它们的应用范围和质量相差很大,

提供辅助的程度也各不相同。

一般而言,测试工具可以分为白盒测试工具、黑盒测试工具、测试定制工具、测试执行工具、测试管理工具和测试支持工具等几大类。

10.2.1 白盒测试工具

白盒测试工具一般是针对被测源程序进行的测试,测试中发现的故障可以定位到代码级,根据测试工具的原理不同,又可以分为静态测试工具和动态测试工具。

1. 静态测试工具

静态测试工具是在不执行程序的情况下,分析软件的特性。静态分析主要集中在需求文档、设计文档以及程序结构上,可以进行类型分析、接口分析、输入输出规格说明分析等。常用的静态分析工具有:McCabe & Associates 公司开发的 McCabe Visual Quality ToolSet 分析工具,ViewLog 公司开发的 LogiScope 分析工具,Software Research 公司开发的 TestWork/Advisor 分析工具及 Software Emancipation 公司开发的 Discover 分析工具等。

按照完成的职能不同,静态测试工具又有以下几种类型:

(1) 代码审查

代码审查工具能帮助人们了解不太熟悉的代码,了解代码的相关性、跟踪程序逻辑、观看程序的图形表达,确认死代码,确定需要特别关照的区域,检查源程序是否遵循了程序设计规则等。这类工具常称为代码审查器。

(2) 一致性检查

一致性检查检测程序的各个单元是否使用了统一的记法或术语,这类工具通常用以检查是否遵循了设计规格说明。此类工具常称为一致性检查器。

(3) 错误检查

错误检查用以确定差异和分析错误的严重性和原因。

(4) 接口分析

接口分析检查程序单元之间接口的一致性,以及是否遵循了预先确定的规则或原则。典型的接口分析包括检查传送给子程序的参数以及检查模块的完整性。此类工具称为接口检查器。

(5) 输入输出规格说明分析

输入输出规格说明分析的目标是借助于分析输入输出规格说明生成测试输入数据。

(6) 数据流分析

数据流分析检测数据的赋值与引用之间是否出现了不合理的现象,如引用未赋值的变量,对以前未曾引用变量的再次赋值等数据流异常现象。

(7) 类型分析

类型分析检测命名的数据项和操作是否得到了正确的使用。通常类型分析用以检测某一实体的值域(或函数等)是否按正确并且一致的形式构成。

(8) 单元分析

单元分析检测单元或构成实体的物理元件是否定义正确和使用一致。

(9) 复杂度分析

有经验的开发人员都知道,80% 的问题是由 20% 的代码引起的。复杂度分析有助于确定分

析域中的风险,帮助测试工程师精确地计划他们的测试活动。换言之,那些标明为较复杂的代码域是必须补充一些测试用例进一步进行审查的域,一般认为是软件测试成本/进度或程序中存在故障的指示器。

2. 动态测试工具

动态测试工具与静态测试工具不同,动态测试工具直接执行被测程序以提供测试支持。它所支持的测试范围十分广泛,包括功能确认与接口测试、覆盖率分析、性能分析、内存分析等。动态测试工具的代表有 Compuware 公司开发的 DevPartner 软件、Rational 公司研制的 Purify 系列。

(1) 功能确认与接口测试

这部分的测试包括对各个模块功能、模块间的接口、局部数据结构、主要执行路径、错误处理等进行测试。

(2) 覆盖分析

一般来说,没有经过测试覆盖分析,软件在发行前仅有 50% 的源程序被测试过。在近一半源代码没有被测试的情况下,大量的故障随软件一起被发行出去,在这种情况下,软件的质量、性能和功能不可能得到保障。此外,什么时候停止测试,是否要对程序作进一步的测试,对于测试工程师和测试管理人员来说是不知道的,通过引进测试覆盖概念,这些问题可以得到解决。

覆盖分析可以对测试质量提供定量的分析。换言之,覆盖分析对所涉及的程序结构元素进行度量,以确定测试执行的充分性。这种测试覆盖分析工具对所有软件测试机构都是必不可少的,它可以告诉被测软件中哪些部分已被测试过,哪些部分还没有被覆盖到,需要进一步测试。

覆盖分析工具大量用于单元测试中。例如,测试对安全性要求较高或与安全有关的系统时,要求达到主要路径覆盖。此外,覆盖分析工具还可以度量设计层次结构如调用树结构的覆盖率。

(3) 性能分析

如果一个应用程序运行缓慢,开发人员很难发现哪里出了问题,程序的性能问题得不到解决,将极大地降低并影响程序的质量,于是查找并修改性能瓶颈已成为改善整个系统性能的关键。

(4) 内存分析

内存泄漏可能会导致系统运行崩溃。内存泄漏是指程序没有释放应该释放的内存单元块,这些内存块从可供分配给所有程序的内存区中“漏”掉了。最后,这种故障将“吃”掉所有的内存,致使程序不能正常运行。如果这种故障出现在资源比较匮乏,应用非常广泛的系统中将可能导致无法预料的重大损失。通过分析内存使用情况,可以了解程序内存分配的真实情况,发现内存的不正常使用,在问题出现前发现征兆,在系统崩溃前发现内存泄露错误,发现内存分配错误,找出发生故障的原因。

10.2.2 黑盒测试工具

黑盒测试是在已知软件产品应具有的功能的条件下,在完全不考虑被测程序内部结构和内部特性的情况下,通过测试来检测每个功能是否都能按照需求规格说明的规定正常工作。黑盒测试工具的代表有 Rational 公司的 TeamTest,Compuware 公司的 QACenter。

常用的黑盒测试工具包括:

(1) 功能测试工具

用于检测被测程序能否达到预期的功能要求并正常运行。

(2) 性能测试工具

性能测试工具有助于确定软件和系统的性能。有些工具还可用于自动多用户客户/服务器加载测试和性能测量,用来生成、控制并分析客户/服务器应用的性能,即性能测试又分为客户端的测试和服务器端的测试。客户端的测试主要关注应用的业务逻辑、用户界面和功能测试等,服务器端的测试主要关注服务器的性能,衡量系统的响应时间,事务处理速度和其他时间敏感等。

10.2.3 测试设计和开发工具

测试设计是说明测试被测软件特征或特征组合的方法,确定并选择相关测试用例的过程。测试开发是将测试设计转换成具体的测试用例的过程。像制定测试计划一样,对最重要、最费脑筋的测试设计过程来说,工具起不了多大的作用,但测试执行和评估类工具,如捕获/回放工具,是有助于测试开发的,也是实施计划和设计合理测试用例的最有效的手段。

测试设计和开发需要的工具类型有:

- 测试数据生成器。
- 基于需求的测试设计工具。
- 捕获/回放。
- 覆盖分析。

测试数据生成工具非常有用,测试数据生成工具可以为被测程序自动生成测试数据,减轻人们在生成大量测试数据时所付出的劳动,同时还可避免测试人员对一部分测试数据的偏见。常用的测试数据生成工具有:Bender & Associates 公司提供的功能测试数据生成工具 SoftTest,Parasoft 公司提供的 C/C ++ 单元测试工具 Parasoft C ++ test 等。其中,路径测试数据生成器是一类常用的测试数据生成工具,但一般程序路径数量太大,路径长度也无限制。多数系统按以下方法选择路径:

- 用户预先指明所有要被分析的路径。
- 用户预先指明循环执行的最大次数及最大路径长度。
- 用户以交互方式选择要被分析的路径,并逐个语句执行这一路径。
- 由系统进行自动选择,以满足测试覆盖的要求。

基于需求的测试设计工具至今还没有获得广泛的实际应用。Aonix 公司提供了一种基于需求和设计的测试数据生成工具 Validator/Req、StP/SE 和 StP/UML。

10.2.4 测试执行和评估工具

测试执行和评估是执行测试用例并对测试结果进行评估的过程,包括选择用于执行的测试用例、设置测试环境、运行所选择的测试、记录测试执行过程、分析潜在的软件故障并测量测试工作的有效性。评估类工具对执行测试用例和评估测试结果这一过程起辅助作用。

测试执行和评估类工具有:

- 捕获/回放。
- 覆盖分析。

- 存储器测试。

对于不断重复执行的测试,可以求助于捕获/回放工具使测试自动化。换言之,就是根据需要,可以几个小时、一个晚上或一天 24 小时不间断地执行测试而不需值班管理。

捕获/回放工具可以捕获用户的操作,包括击键、鼠标活动,并显示输出。这些被捕获的测试,包括已被测试人员确认的输出,需要时,工具可以自动回放以前捕获的测试,并通过与以前存储的结果进行比较而对结果进行确认。因此,当故障修复或为增强软件功能而进行修改时,测试人员不需要通过手工反复不断地重新执行测试。

存储器测试类工具一般来说能检测:

- 存储器问题。
- 重写或重读存储阵列边界。
- 已分配但未释放的内存。
- 读出并使用未初始化的存储器。

利用存储器测试工具可以在故障发生之前将其确认。详细的诊断信息可以跟踪并消除故障。存储器测试工具大多是语言专用和平台专用的,有些可用于最常见的环境。这方面最好的工具是非侵入式的,且使用方便、价格合理。

10.2.5 测试管理工具

测试管理工具是指帮助完成制定测试计划,跟踪测试运行结果等的工具。一个小型软件项目可能有数千个测试用例要执行,使用捕获/回放工具可以建立测试并使其自动执行,但仍需要测试管理工具对成千上万个杂乱无章的测试用例进行管理。

测试管理工具用于对测试进行管理。一般而言,测试管理工具对测试计划、测试用例、测试实施进行管理,还包括缺陷跟踪管理工具等。测试管理工具的代表有 Rational 公司的 Test Manager,Compureware 公司的 TrackRecord 等。测试管理工具包括以下内容。

(1) 测试用例管理

最好的测试用例管理工具具备以下一些功能:

- 提供用户界面用于管理测试。
- 对测试进行整理以方便使用和维护。
- 启动并管理测试执行,运行用户选择的测试。
- 提供与捕获/回放及覆盖分析工具的无缝集成。
- 提供自动化的测试报告和相关文档编制。

(2) 缺陷跟踪管理

缺陷管理工具有时又称为问题跟踪工具、故障管理工具等,用于在整个软件生存周期中对缺陷进行跟踪管理和强化管理记录、跟踪并提供全面的帮助。最好的缺陷管理工具很容易根据特定的环境进行定制,并具备以下一些特征:

- 能十分容易并迅速地提交和更新故障报告。
- 能十分容易地生成预先定义或用户定义的管理报告。
- 能十分容易并有选择地自动通知用户对故障状态的修改。
- 很容易根据用户提问提供对所有数据的安全访问。

(3) 配置管理

配置管理是对文档以及其他紧要事物进行管理、控制和协调的关键。配置管理工具协助版本控制并构建管理过程。

10.2.6 测试工具的选择

面对如此众多的测试工具,对工具的选择就成了一个比较重要的问题。在考虑选用工具的时候,建议从以下几个方面来权衡和选择。

1. 功能

功能当然是最关注的内容,选择一个测试工具首先就是看它提供的功能,但这并不是说测试工具提供的功能越多就越好,适用才是根本。为不需要的功能花费金钱实在是不明智的行为。事实上,目前市场上同类软件测试工具的基本功能大同小异,只不过侧重点有所不同而已。例如,同为白盒测试工具的 Logiscope 和 PRQA 软件,他们提供的基本功能大致相同,只是在编码规则、编码规则的定制、采用的代码质量标准方面有所不同。除了基本的功能之外,选择测试工具时,也可以参考下面的功能需求。

① 报表功能。测试工具生成的结果最终由人来进行解释,查看最终报告的人不一定对测试熟悉,因此,测试工具能否生成结果报表,以什么形式提供报表是需要考虑的因素之一。

② 测试工具的集成能力。测试工具的引入是一个伴随测试过程改进而进行的长期过程,因此,测试工具的集成能力也是必须考虑的因素,这里的集成包括两方面的含义:

- 测试工具能否和开发工具进行良好的集成。
- 测试工具能否和其他测试工具进行良好的集成。

③ 和操作系统及开发工具的兼容性。测试工具是否可以跨平台,是否适用于公司目前使用的开发工具,这些问题也是选择一个测试工具时应该考虑的问题。

2. 成本

选择工具时应该进行成本/收益分析。工具销售商往往热衷于介绍他们的工具能做什么,如何能解决具体问题。我们所关心的是成本有多高。产品价格是最基本的成本,此外还有附加成本,包括挑选、安装、运输、培训、维护和支持,以及改组过程等,重要的是确定实际成本——总成本,甚至生命期成本。

与需要更换测试过程相比,支持已有测试过程的工具在人力和财力方面实施起来相对比较容易。市面上有好几百种测试工具,开发公司在规模、已建立的客户库、产品成熟度、管理深度以及对测试和工具的理解方面差别很大。为高效率地实施工程项目,选择测试工具之前,最好考虑一下下面的问题:

- 工具怎样介入并支持测试过程。
- 知道怎样计划并设计测试。

毫无疑问,使用工具可以使测试更容易、更有效、更高产,但工具不能代替思考、计划和设计。应尽量避免将昂贵的测试工具束之高阁。

10.3 常用测试工具简介

10.3.1 Parasoft C ++ Test 测试工具简介

Parasoft C ++ Test 是 Parasoft 公司开发的一个针对 C/C ++ 源程序代码进行自动单元测试的工具。它可以对被测程序进行白盒测试、黑盒测试以及回归测试。

(1) 白盒测试

Parasoft C ++ Test 对 C/C ++ 被测源程序进行分析,对所有类的成员函数即公共成员函数、保护的成员函数以及私有成员函数进行测试。(Record 命令)

进行白盒测试时,Parasoft C ++ Test 对指定的文件、类或者函数自动生成测试用例,当输入一个非法参数时,判断有关函数是否能够正确处理等。

(2) 黑盒测试

Parasoft C ++ Test 不对被测源程序进行分析,只对类的公共接口函数进行测试。(Play 命令)

黑盒测试时,Parasoft C ++ Test 不自动生成测试用例,而是直接运行在‘ 测试用例编辑器 ”中当前已有的测试用例(手工添加的)。

(3) 回归测试

在修改被测源程序后,用原有的测试用例进行重新测试。(Play 命令)

实际使用时,建议首先使用 Record 命令执行一次白盒测试,以便 Parasoft C ++ Test 根据类的成员函数自动生成相应的测试用例,然后再根据需要手工添加一些测试用例,最后再通过 Play 命令执行一次黑盒测试。

如果被测源程序没有对各种可能情况尤其是边界情况进行特殊的处理,Parasoft C ++ Test 可以发现这些潜在的问题。对于一些特殊的测试情况,还可以手工设计一些测试用例。此外,采用 Parasoft C ++ Test 还可以帮助人们检查程序的编码错误,判断被测源程序是否严格按编码规范进行开发等。

Parasoft C ++ Test 使用比较简单,既可以针对一个 VC 工程进行全面的测试,也可以只对一个 C/C ++ 源文件进行测试。但是,当项目比较大时,最好按文件一个一个地测试,不要直接对一个工程进行自动测试。

10.3.2 白盒工具——NuMega DecPartner Studio

NuMega DecPartner Studio 是 Compuware 公司开发的一组白盒测试工具套件,主要用于代码开发阶段,检查代码的可靠性和稳定性,它提供了先进的错误检查和调试解决方案,可以充分改善软件的质量。

NuMega DecPartner Studio 工具有自动故障检查、性能测试、代码覆盖分析等功能,主要有 3 个独立的模块:

- BoundsChecker
- TrueCoverage
- TrueTime

1. BoundsChecker

在开发过程中经常遇到这种情况 程序语法没有问题,代码也没有错误,但程序运行就是不正确甚至出现死机。这种现象很可能是由逻辑错误引起的内存溢出或资源泄露等问题引起的,但这种故障一般不容易被检测出来的。BoundsChecker 是一个非常有效的故障检测工具,主要检测程序运行时的各种故障。

通过对被测程序进行测试,BoundsChecker 能够提供清晰的、详细的故障分析,自动查明静态堆栈错误以及内存/资源泄露等问题,并能迅速地定位出错的源代码,即使在没有源代码的情况下也可以检查出第三方组件的错误。

BoundsChecker 能检测的故障包括:

(1) 指针和泄露故障

- 接口泄露。
- 内存泄露。
- 资源泄露。
- 指针操作故障。

(2) 内存错误

- 动态存储溢出。
- 内存分配冲突。
- 栈空间溢出。
- 静态存储溢出。
- 内存操作溢出。

(3) API 和 OLE 故障。

- API 函数返回失败。
- API 函数未执行。
- 无效的变量。
- OLE 接口方法失败。
- 线程调用函数故障。

BoundsChecker 支持的语言和主机平台为:

C ++、Delphi ;Windows NT、Windows 95/98。

2. TrueCoverage

TrueCoverages 是一个代码覆盖分析工具。开发过程中,对一个被测程序进行手工测试,总会有一部分代码功能没有被检测到,或者是由于逐个检测每一个函数调用需要太多的时间,没有足够的时间了,对于未被测试的代码,不能保证其正确性,但程序的失效往往是由这部分未测试的代码造成的。TrueCoverage 可以帮助测试人员解决这些问题,在测试程序的过程中,每执行一次被测程序,TrueCoverage 给出这次执行中所有函数被调用的次数、所占的比率等,并可直接定位到源代码。TrueCoverage 能通过衡量和跟踪代码执行,帮助开发人员节省时间和改善代码的可靠性。

TrueCoverage 支持的语言和主机平台为:

C ++、JAVA、Visual Basic ;Windows NT、Windows 95/98。

3. TrueTime

如何提高代码的执行效率,是开发过程中一个重要问题。一个系统执行速度较慢时,系统的性能将受到影响。如何快速地查找出性能瓶颈呢?TrueTime 就是一个对系统运行性能进行分析的工具。

TrueTime 能够收集系统运行时性能方面的相关数据,包括每个模块、每个函数的运行性能。对于源程序代码,TrueTime 还可以给出每一行代码的运行性能,通过这些数据,可以帮助开发人员确定系统的性能瓶颈所在,根据需要来优化系统的性能。

TrueTime 支持的语言和主机平台为:

C ++、JAVA、Visual Basic;Windows NT、Windows 95/98。

4. SmartCheck

在 Visual Basic 程序的开发过程中,经常会遇到许多问题难以解决,比如像隐藏的 Run – time 错误、组件错误等,这些错误很难定位到具体的代码中,开发人员需要花费大量时间去寻找和解决。SmartCheck 是一个能很快找到这些问题的自动化工具,对于 Visual Basic 来说是最好的 Run – time 调试工具,可以检测所有的 Windows API 函数调用、内存分配以及其他一些重要的软件故障。SmartCheck 查错的种类包括泄露、接口方法失败、存储错误、程序和函数失败以及 Run – time 错误等,并能将检测到的错误快速地定位到源代码。使用 SmartCheck 将会极大地提高 VB 开发人员的工作效率。

SmartCheck 支持的语言和主机平台为:

Visual Basic;Windows NT、Windows 95/98。

5. FailSafe

FailSafe 是用于 Visual Basic 开发的一个自动错误处理和恢复系统。VB 开发人员经常遇到程序执行时意外地终止了,但对于为什么会出现这种情况只提供了一些简单、模糊的出错信息,开发人员不能很快发现问题的根源。如果使用了 FailSafe,它通过程序插装技术在被测程序中插装一些额外的代码,当程序执行时,FailSafe 通过这些插入的代码捕获、记录执行时程序和系统的重要信息,直接指出故障发生时程序和系统的状态,这些信息可以帮助开发人员快速且准确地解决问题。

FailSafe 支持的语言和主机平台为:

Visual Basic;Windows NT、Windows 95/98。

6. CodeReview

CodeReview 是一个最好的源代码自动分析工具,它对被测程序的组件、逻辑、Windows 和 VB 自身潜在的数百个问题进行严格的代码检查。CodeReview 分析的类型包括 Y2K 问题、逻辑错误、应用程序性能和可用性问题,Windows API 调用和标准一致性问题等,还可以对整个 VB 工程或指定的模块进行检测,并能定制查错的种类。对检测的结果有详细的说明,能够提供帮助和推荐解决方案,而且能够直接链接到源代码上。

CodeReview 支持的语言和主机平台为:

Visual Basic;Windows NT、Windows 95/98。

7. JCheck

JCheck 对于 Java 开发人员来说是一个功能强大的图形化线程和事件分析工具,它提供了一

个生动的图形化方法来表示程序的线程状态信息以及和 Windows 线程、同步对象、线程组等进行交互的状态信息,使开发人员能够直观地分析 Java Applet 或 Application。通过这些形象化的图形显示,可以确定 Run-time 错误,对逻辑错误进行分析,发现线程问题如死锁、资源缺乏和系统失效等,诊断线程同步和时间选择问题,分析程序执行流程等。JCheck 对于这些故障可以定位到源代码并显示详细的信息,极大地减少了程序的调试时间,改善了软件开发的效率。

JCheck 支持的语言和主机平台为:

Microsoft Visual J ++ ;Windows NT、Windows 95/98。

10.3.3 黑盒测试工具——QACenter

QACenter 测试工具能够自动地帮助管理测试过程,快速分析和调试程序,能够针对回归测试、强度测试、单元测试、并发测试、集成测试、移植测试、容量和负载测试建立测试用例,自动执行测试并产生相应的文档。

QACenter 主要包括以下几个模块。

- QARun 功能测试工具。
- QALoad 性能测试工具。
- Eco Tools 可用性管理工具。
- EcoScope 性能优化工具。

1. 功能测试工具 QARun

在 QACenter 测试工具套件中,QARun 模块主要用于客户/服务器系统中客户端的功能测试。在功能测试中,主要包括对系统的 GUI(图形用户界面)进行测试以及对客户端事务逻辑进行测试。不断变化的需求将导致不同版本的软件,每一个版本都需要进行测试,而且每一个被修改的地方往往是最容易发生故障的地方,回归测试是测试中最重要的测试。QARun 可以为回归测试提供支持。

QARun 的测试实现方式是通过鼠标移动、键盘点击操作被测系统,进而得到相应的测试脚本,对该脚本进行编辑和调试。在记录过程中针对被测系统中所包含的功能点进行基线值的建立,换句话说就是在插入检查点的同时建立期望输出值。通常,检查点在 QARun 提示目标系统执行一系列事件之后被执行,检查点可以确定实际结果与期望结果是否相同。

2. 性能测试工具 QALoad

QALoad 是一个负载测试工具,该工具支持的范围广、测试的内容多,可以帮助软件测试人员,开发人员和系统管理人员对于分布式的被测程序进行有效的负载测试。负载测试能够模拟大批量用户的活动,从而发现在大量用户负载下对 C/S 系统的影响。

测试人员只需操作被测系统,执行关键性能的事务处理,然后在 QALoad 脚本中通过服务器调用系统的需求类型,开发相应的事务处理,创建完整的功能脚本。QALoad 的测试脚本开发由捕获会话、转换捕获会话到脚本以及修改和编译脚本等一系列过程组成。一旦脚本编译通过后,使用 QALoad 的机构把脚本分配到测试环境的相应机器上,驱动多个 play agent 模拟大量用户的并发操作,实施系统的负载测试,减轻了以往大量的人工工作,节省了时间,提高了效率。

3. 可用性管理工具 EcoTools

性能测试完成之后,应对系统的可用性进行分析。很多因素影响系统的可用性,用户的桌

面、网络、服务器、数据库环境以及各式各样的子组件都可以链接在一起,任何一个组件都可能造成整个系统对最终用户不可使用。

EcoTools 工具提供了一个打包的 Agent 和 Scenarios,可以在测试或生产环境中激活,计划和管理以商务为中心的系统的可用性。QALoad 对于在服务器上设置加载测试是一个很好的工具,但不能承担诊断问题的工作。而 QALoad 与 EcoTools 集成可以为所有加载测试和计划项目需求能力提供全方位的解决方案,允许在图形中查看 EcoTools 资源利用率。

EcoTools 工具包括数百个 Agents,可以监控服务器资源,尤其是监控 Windows NT、UNIX 系统、Oracle、Sybase、SQL Server 和其他应用包。

4. 性能优化工具 EcoScope

EcoScope 是一套定位于应用系统及其所依赖的网络资源的性能优化工具。EcoScope 可以提供应用视图,并给出应用系统是如何与基础架构相关联的。这种视图是其他网络管理工具所不能提供的。EcoScope 能解决在大型企业复杂环境下分析与测量应用系统性能的难题。通过提供应用的性能级别及其支撑架构的信息,EcoScope 能帮助 IT 部门就如何提高应用系统的性能提出多方面的决策方案。

EcoScope 使用综合软件探测技术无干扰地监控网络,可以自动跟踪在 LAN/WAN 上的应用流量、采集详细的性能指标,并将这些信息关联到一个交互式的用户界面中,自动识别低性能的应用系统、受影响的服务器与用户性能低下的程度。用户界面允许以一种智能方式访问大量的 EcoScope 数据,所以能很快地找到性能问题的根源,并在几小时内解决令人烦恼的性能问题,而不是几周甚至几个月。

10.3.4 数据库测试工具

1. 数据库测试数据自动生成工具——TESTBytes

在数据库开发过程中,为了测试应用程序对数据库的访问,需要在数据库中生成测试数据。当数据库中只有少量的数据时,程序可能没有问题,但在真正投入运用时,因为实际数据库有大量的数据,问题可能就出现了,所以应该尽早地通过在数据库中生成大量数据来帮助开发人员尽快完善这部分功能和性能。

TESTBytes 是一个用于自动生成测试数据的易用的工具,通过简单的点击式操作,就可以确定要生成的数据类型(包括特殊字符的定制),并通过与数据库的连接自动生成数百万个测试数据,可以极大地提高数据库开发人员、软件质量保证人员、测试人员、数据仓库开发人员、应用开发人员的工作效率。

TESTBytes 支持的平台为:

Windows NT、Windows 95/98、Windows 3.x

2. 数据库测试工具——ERwin Examiner

ERwin Examiner 通过检测数据库脚本文件、ERwin 建模文件或直接连接现有的数据库进行检测,并针对数据库中的字段、索引、约束、规范、关系等数据库属性生成检测报告。ERwin Examiner 能够分析数据模型,找到那些可能会破坏数据库完整性和功效的矛盾之处,可以调整用户的数据库设计,自动生成修改脚本以修改设计,从而加快数据库的设计与部署。

ERwin Examiner 的主要特征包括:

• 综合分析与报告 ERwin Examiner 完善的分析功能能够在相关模型和特定数据库性能参数基础上生成详细的分析报告,清晰地列出了栏目、索引、约束条件、规范和关联等属性,精确地指出数据库设计中的矛盾之处或异常之处。

• 自动改进数据库设计 ERwin Examiner 详细的分析报告有助于加快数据库模型的验证,在综合分析结果的基础之上,可以提出对数据库模型的改进意见,进而高效一致地改进数据库设计。

• Teach Me 功能 除了提供特定的分析信息与建议外,ERwin Examiner 的 Teach me 功能还可以根据关系原理揭示设计或更改所造成的影响,为建模人员指出设计决策的结果,从而帮助他们建立更高质量的数据库。

10.3.5 测试管理工具——TestDirector

MI 公司开发的测试管理工具 TestDirector,不仅可以用于对白盒测试进行管理,而且还可以用于对黑盒测试进行管理,它是一个基于 Web 的测试管理工具,这就意味着用户可以通过局域网或 Internet 来访问 TestDirector 工具,并将使用管理工具的对象从项目管理人员扩大到了软件质量保证部门、用户和其他相关的部门,这是以往测试管理工具做不到的。

TestDirector 包括以下 4 个主要部分:

- 需求分析。
- 测试计划。
- 运行。
- 缺陷管理。

这 4 部分可以有效地控制需求分析覆盖、测试计划管理、自动化测试脚本运行和对测试中发现的故障进行跟踪,并确保故障有相应的人员进行修复或制作相应的补丁。当然,对人员的权限授予和控制访问是默认项,是每个测试管理工具都具有的,TestDirector 也不例外。

通过 TestDirector 测试管理工具件,用户可以及时地掌握软件的测试和完成情况,并对整个过程进行监督和管理,有利于对成本和时间进行有效的管理。

10.4 测试自动化和测试工具的好处

测试自动化和测试工具不仅可以提高测试任务执行的效率,还有助于:

① 对新版本进行回归测试。对于产品型的软件,新版本的大部分功能及界面都和上一个版本相似或者完全相同,这部分功能特别适合于自动化测试。回归测试是测试自动化的强项,它能够很好的测试新版本是否引入了新的故障,老的故障是否修改过来了。

② 执行更多更频繁的测试。对于多次重复、机械性的动作,比如要向系统输入大量的相似数据来测试压力和报表,人工测试非常的耗时和繁琐,测试工具的一个显而易见的好处是可以在较少的时间内运行更多的测试。

③ 执行一些手工测试困难或不可能做的测试。例如,对于 300 个用户的联机系统,用手工进行压力测试、并发操作的测试几乎是不可能的,但自动测试工具可以模拟来自 300 个用户的输入。

④ 更好地利用资源。理想的自动化测试能够按计划完全自动地运行,可以充分利用了资源,在周末和晚上执行自动测试,避免了开发和测试之间的等待。将烦琐的任务自动化,如重复输入相同的测试输入,可以提高准确性和测试人员的积极性,将测试人员解脱出来投入更多精力设计更好的测试用例。

⑤ 测试具有一致性和可重复性。由于每次自动测试工具运行的脚本都相同的,所以每次执行的测试具有一致性,这在手工测试中是很难保证的。利用自动化测试的一致性,可以很容易地发现被测软件的任何改变。

有些测试可能在不同的硬件配置下执行,使用不同的操作系统或不同的数据库,此时要求跨平台质量的一致性,这在手工测试情况下更不可能做到。

⑥ 测试的复用性。自动测试重用的次数比手工重复相同测试的次数要多得多。

⑦ 增加软件信任度。一旦得知软件通过强有力的自动测试后,软件发布时对其的信任度也高。

⑧ 可以更快地将软件推向市场。一旦一系列测试已经被自动化,则可以比手工测试更快地重复执行,因此缩短了测试时间。

总之,测试自动化和测试工具能够通过较少的开销可以获得更彻底的测试,提高软件产品的质量。

10.5 测试自动化和测试工具存在的问题

使用自动测试可能会遇到许多问题。下面是一些普遍存在的问题。

① 不现实的期望。容易对新工具持乐观态度,期望这工具可以解决目前遇到的所有问题。

② 缺乏测试实践经验。如果缺乏测试实践经验,测试发现故障的能力较差,在这种情况下采用自动测试工具并不是一个好办法。改进测试的有效性比改进测试的效率要好得多。

③ 期望自动测试工具能取代手工测试。不可能也不能期望将所有测试都自动化。下列一些情况更适合进行手工测试:

- 测试很少运行。例如,一年只运行一次,这种情况下不值得将测试自动化。
- 软件不稳定。例如,如果软件从一个版本到另一个版本,在这期间用户界面和功能频繁变化,那么修改相应的自动化测试开销较大。
- 结果很容易由人来验证,但测试自动化实现很困难甚至是不可能的。例如,彩色模式的合适程度,屏幕轮廓的直观效果或选择指定的屏幕对象是否能播放正确的声音等。
- 涉及物理交互的测试,例如在读卡机上划片,断开某些设备的连接,开关电源等。

④ 期望自动测试发现新故障。不要期望自动测试能发现许多新的故障。测试在首次运行时最有可能发现故障,以后再运行相同的测试发现新故障的可能性很小。

测试执行工具是一种回归测试工具,用于重复已经执行过的测试。因此,不能用来发现大量新的故障,特别是运行在与以前相同的硬件和软件环境中。发现新故障应该是手工测试的主要目的。测试专家 James Bach 总结得出:85% 的故障靠手工发现,而自动化测试只能发现 15% 的故障。

⑤ 安全性错觉。没有发现任何故障并不意味软件没有故障,可能是测试不全面或测试本身

就有故障。测试自动化工具只能判断实际结果和期望结果之间的差别。如果自动测试报告通过所有测试,实际上只能说明实际结果与期望结果匹配。

⑥ 测试自动化不能提高有效性。自动化测试并不会比手工运行相同的测试更加有效,自动化只能提高测试的效率,即运行测试的开销和时间。

⑦ 自动测试的维护性。软件修改后,经常需要修改部分或全部测试,以便可以重新正确地运行,自动测试也是如此。当修改测试的费用比手工重新测试更高时,测试自动化将被放弃。

⑧ 测试自动化可能会制约软件开发。自动测试可能比手工测试更‘脆弱”。软件部分改变有可能使自动测试软件崩溃。由于经济原因,对自动测试影响较大的软件修改可能受到限制。

⑨ 工具本身没有想像力。工具毕竟是工具,人感官方面的东西,比如界面的美观、声音的体验、易用性等,只有靠人来测试,自动测试工具是无能为力的。测试工具与其他软件的互操作性,也是一个严重的问题。此外,人工测试可以处理意外事件,例如,网络连接中断等。工具可以具有处理某些事件的功能,但毕竟不如人灵活。

⑩ 组织问题。自动测试实施起来并不简单。自动化测试的推行,有很多阻力,比如机构是否重视,是否有这样的技术水平,由于测试脚本的维护工作量较大的,是否值得维护等问题都必须考虑。

测试自动化不仅仅只与项目有关,在大型组织中,测试自动化很少根据一个项目进行评价,因为所有项目都可能面临许多问题而使收效甚微。应有标准确保测试机构中使用工具的一致性,否则每个小组开发自己的测试自动化方法,这样在测试人员之间很难互通或共享自动测试。

小结

软件测试自动化可以省去许多繁杂的工作,节省软件测试时间,提供比手工测试更好、更快的测试执行方式。但测试自动化的目的在于发现老的软件故障,而手工测试的目的在于发现新故障。

测试工具可以分为白盒测试工具、黑盒测试工具、测试制定工具、测试执行工具、测试管理工具和测试支持工具等几类。

测试自动化和测试工具能够通过较少的开销获得更彻底的测试,来提高软件产品的质量。但使用自动测试时,也会遇到许多问题,因为工具毕竟是工具,在处理一些意外事件时,不如人灵活。

第 10 章习题

1. 给出使用软件测试工具和测试自动化的一些好处。
2. 在决定使用测试工具和自动化时,要注意哪些缺点或事项?
3. 主要的测试工具可以分为几大类?

第 11 章　软件质量保证

随着软件开发规模的加大、复杂程度的增加,以寻找软件故障为目的的测试工作就显得更加困难。然而,为了尽可能多地找出软件中存在的故障,开发出高质量的软件产品,加强对测试工作的组织和管理就显得尤为重要。建立软件测试管理体系的主要目的是确保软件测试在软件质量保证中发挥应有的关键作用,而测试机构的结构在一定程度上是解决问题的最有效的方法,即可以有效地协调好人们之间的相互关系。

本章重点:

- 软件测试质量保证
- 软件测试管理技术
- 测试的组织形式
- 软件过程成熟度

11.1　软件质量保证

测试能帮助确保一个软件产品满足需求,但测试并不是质量保证。有些人错误地将测试和质量保证等同起来。在许多机构中,软件质量保证部门通常负责开发测试计划和执行系统测试软件质量保证,可能也会对开发过程中的测试进行监测和保管统计数据。软件质量保证部门从事的是那些用来防止和去除软件故障的活动,他们负责制订为了开发出更好的软件应该遵守的标准,包括定义为理解设计意图而创建的各种文档的类型,指导项目活动的过程以及量化决议结果的方法。

测试将提高计算机系统的质量。测试可以发现故障,从而帮助开发者发现问题并纠正问题。测试是任何质量保证过程中必需的但不是所有的部分。对一个系统测试得越多,就越能确保它的正确性,然而测试通常不能保证系统运转百分之百正确。因此,测试在确保质量方面的主要贡献在于识别那些在一开始就应该能够避免的错误。软件质量保证的使命首先是避免错误,要做到这一点,除了测试外还需要其他方面的处理。

一个软件产品或系统的质量如何,是从设计开始就决定了的,它和整个软件的研制过程密切相关。Croby 认为 制造高质量的软件产品并不比生产低质量的产品需要更多的费用。他将质量费用分为两类 整合费用和非整合费用。

(1) 整合费用

整合费用是指与一次性计划和执行测试相关的全部费用,用于保证软件按照预期方式运行。

(2) 非整合费用

如果发现了软件故障,必须花时间分离、报告和进行回归测试以保证其得以修复,所有这些花费称为非整合费用。如果软件故障在发布之前发现,解决这些问题的费用属于内部非整合费用。如果软件故障被遗漏并落到用户手里,结果就是代价高昂的产品支持电话,可能还需要修

复、重新测试和发布软件。更糟糕的情况是,产品被召回或者卷入官司。解决这些问题的所有费用属于外部非整合费用。

Crosby 说明了软件发布前的整合费用加内部非整合费用小于外部非整合费用。也就是说,尽早剔除软件故障,或者在理想情况下一开始就没有故障,产品的开销就会比其他情况小。美国质量保证研究所对软件测试的研究结果也表明:越早发现软件中存在的问题,开发费用就越低,在编码后修改软件故障的成本是编码前的 10 倍,在产品交付后修改软件故障的成本是交付前的 10 倍,软件质量越高,软件发布后的维护费用越低。

软件的质量不仅体现在程序的正确性上,它和编码以前所做的需求分析,软件设计等密切相关。软件使用中出现的故障,不一定是编程人员在代码编写阶段引入的,很可能在软件设计,甚而需求分析阶段就埋下了祸根。这时,对故障的纠正不能采取简单的修修补补,而必须追溯到软件开发的最初阶段,这无疑加大了软件的开发费用。因此,为了保证软件的质量,应该着眼于整个软件生存期,特别是着眼于编码以前的各开发阶段的工作。软件质量保证包括以下多方面的工作:

- 采用技术手段。
- 组织正式评审。
- 软件测试。
- 推行软件工程标准。
- 对软件的变更进行控制。
- 对软件质量进行度量。
- 对软件质量情况及时记录和报告。

软件质量保证的主要职责是检查和评价当前软件开发过程,并设法达到防止软件故障出现的目标。

在软件规格说明和设计资料完成以后,就要对它们的质量进行评价,其目的在于揭露软件的质量问题。软件开发人员把软件测试当做质量保证的重要手段,测试可以发现软件的大多数隐藏的故障,遗憾的是测试并不能发现所有的软件故障。

软件质量的一个不可忽视的威胁因素来自软件的修改和变更。尽管表面上看起来,修改和变更是有理由和有益的,但实践证明,在修改过程中常常引进一些潜伏的故障,或是带来一些足以传播故障的因素。因此,严格控制软件的修改和变更自然是十分必要的,包括严格掌握修改和变更请求,仔细研究修改和变更的性质以及控制修改和变更对软件各部分和有关方面引起的冲击,这是软件维护工作的一个重要问题。

11.2 软件测试管理技术

软件测试是软件质量保证的关键步骤。根据对国际著名 IT 企业的统计,它们的软件测试费用占整个软件工程所有研发费用的 50% 以上。相比之下,我国软件企业在软件测试方面与国际水准差距较大。大多数企业在管理上随意、简单,没有建立有效、规范的软件测试管理体系。

应用系统方法来建立软件测试管理体系,也就是把测试工作作为一个系统,对组成这个系统的各个过程加以识别和管理,以实现特定的系统目标,同时要使这些过程协同作用、互相促进,尽

可能发现并排除软件故障。测试系统主要由下面 6 个相互关联、相互作用的过程组成。

(1) 测试计划

确定各测试阶段的目标和策略。这个过程将输出测试计划,明确要完成的测试活动,评估完成活动所需要的时间和资源,进行活动的安排和资源分配等。制定测试计划与软件开发同步进行,在需求分析阶段,要制定验收测试计划,并与需求规格说明一起提交评审。类似地,在概要设计阶段,要制定系统测试计划,在详细设计阶段,要制定和集成测试计划,在编码实现阶段,要制定单元测试计划。对于测试计划的修订部分,需要进行重新评审。

(2) 测试设计

根据测试计划设计测试方案。测试设计过程输出的是各测试阶段使用的测试用例。测试设计也与软件开发同步进行,其结果可以作为各阶段测试计划的附件提交评审。测试设计的另一项内容是进行回归测试设计,即确定回归测试用例集。对于测试用例的修订部分,也要求进行重新评审。

(3) 测试实施

使用测试用例运行程序,将获得的运行结果与预期结果进行比较和分析,记录、跟踪和管理软件故障,最终得到测试报告。

(4) 配置管理

测试配置管理是软件配置管理的子集,作用于测试的各个阶段,其管理对象包括测试计划、测试用例、被测版本、测试工具及环境、测试结果等。

(5) 资源管理

包括对人力资源和工作场所以及相关设施和技术支持的管理。如果建立了测试实验室,还存在其他的管理问题。

(6) 测试管理

采用适宜的方法对上述过程及结果进行监视,并在适当的时候进行测量,以保证上述过程的有效性。如果没有实现预定的结果,则应进行适当的调整或纠正。

此外,测试与软件修改过程相互关联、相互作用。测试输出的软件故障报告是软件修改的输入。反过来,软件修改输出新的被测版本又成为测试的输入。根据上述 6 个过程,可以确定建立软件测试管理体系的 6 个步骤:

① 识别软件测试所需要的过程及其应用,即测试规划、测试设计、测试实施、配置管理、资源管理和测试管理。

② 确定这些过程的顺序和相互作用,前一过程的输出是后一过程的输入。其中,配置管理和资源管理是这些过程的支持性过程,测试管理则对其他测试过程进行监视和管理。

③ 确定这些过程所需要的准则和方法,一般应制订这些过程形成的文档,以及监视、测量和控制这些过程的准则和方法。

④ 确保可以获得必要的资源和信息,以支持这些过程的运行和对它们的监测。

⑤ 监视、测量和分析这些过程。

⑥ 实施必要的改进措施。

软件过程的监视和测量是对软件产品的特性进行监视和测量,主要依据是软件需求规格说明,验证产品是否满足要求,所开发的软件只有符合预先设定的指标,才可以交付。对于软件测

试中发现的软件故障,要认真记录它们的属性和处理措施,并进行跟踪,直至最终解决。在排除软件故障之后,需再次进行验证。通过设计测试用例对需求分析、软件设计、程序代码进行验证,确保程序代码与软件设计说明一致,软件设计说明与需求规格说明的一致。对于验证中发现的不合理现象,同样要认真记录和处理,并跟踪解决。解决之后,也要再次进行验证。软件过程的监视和测量可以从软件测试中获取大量的数据和信息,用于判断这些过程的有效性,为软件过程的正常运行和持续改进提供决策依据。

建立软件测试管理体系的主要目的是确保软件测试在软件质量保证中发挥应有的关键作用。

11.3 测试的组织方式

测试小组是专门从事测试活动的资源或一系列的资源。随着公司的扩大,必须要有专门、独立的测试机构。只有不持偏见的人才能提供不持偏见的评价,测试要在度量软件质量方面真正有效果,就必须独立进行。

① 测试管理很困难,测试小组的管理人员必须具备:

- 理解并具有评价软件测试过程、标准、策略、工具、培训和度量的能力;
- 维护一个测试机构的能力,该机构必须坚强有力、独立自主、办事正规并且没有偏见;
- 领导、交流、支持及控制的能力;
- 关照测试小组的时间。

② 测试机构改组时,必须充分考虑到对测试的影响,不当的软件测试机构可能会带来一些危险,包括:

- 测试的独立性、正规性和思想倾向会被削弱或消除。
- 测试人员不参与有回报的项目。
- 测试人员不够。

③ 测试人员结构不合理,初级人员太多。

- 测试的管理人员级别太低。
- 没有测试知识、培训、工具和过程方面的优势。
- 测试没有能力阻止发送质量低下的产品。
- 缺乏集中、持续的改进过程。
- 管理人员缺乏管理测试小组的能力。
- 不强调质量问题。
- 测试管理人员由于没有获得事业上的进步而沮丧。

制订改组计划时,上述各项可以用做需要考虑问题的清单。此外,测试小组可用的组织结构很多,各有其优势和不足。下面是一些常见的组织结构。

图11-1显示了小型(小于10人)开发小组常用的结构。在该结构中,开发管理员同时管理测试人员和开发人员的工作。编写代码的人和在代码中寻找故障的人向同一个人报告,这就有可能引起问题。开发管理员除指导产品开发小组外,还必须就测试过程、测试标准、测试策略、测试工具、测试培训、测试度量、专业测试人员的聘用等方面提供指导。这样一来,可能什么事情

办不成。

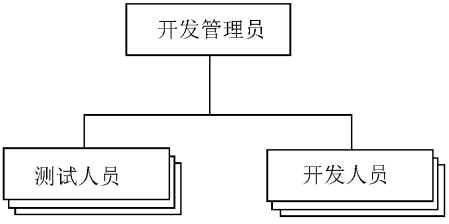


图 11－1 测试小组向开发管理员报告

再者,利益冲突在所难免。开发管理员的目标是促使其小组开发软件,测试人员报告软件故障只会妨碍这个过程。测试人员这边工作得很好,而开发人员那边却情况不妙。如果管理员为测试人员提供更多资源和经费,他们会找出更多软件故障,但是他们找出的软件故障越多,就越妨碍管理员实现其软件的目标。

很明显,必须成立单独的测试小组——专门从事测试活动的一套资源,由一位测试经理管理,该组织可以连续为所有的项目服务——为管理层提供独立、不带偏见的高质量的信息。

Bill Hetzel 在其《软件测试指南大全》一书中指出,独立的测试机构十分重要,主要原因有:

- 没有这样一个机构,建立的系统就不会理想。
- 有效的度量对于产品质量控制是十分重要的。
- 测试协调需要全职、专门的人员投入。

图 11－2 给出了另一种常用的组织结构,测试团队和开发团队都向项目管理员报告。在该组织方式下,测试团队一般有自己的负责人或者管理员,其利益和注意力集中在测试小组及其工作上。这种独立性对软件质量做重大决定时极有好处。测试小组的意见和开发人员以及其他参与产品制作者的意见同等重要。

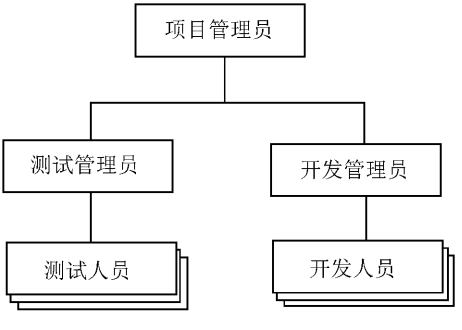


图 11－2 测试小组和开发小组相对独立

然而,该组织结构的缺点是项目管理员对质量进行最终决定。这也许没问题,不少行业容易接受。但是在高风险或者任务要求严格的系统开发中,关于质量的意见拥有更高的等级有时是有益的。图 11－3 所示的组织方式代表此类结构。在该组织方式中,负责软件质量的小组直接向高级管理员报告。该团队持有的独立性允许他们建立自己的标准和规范,评价结果,采取跨越

多个项目的处理措施。关于质量优劣的信息可以直达最高层。

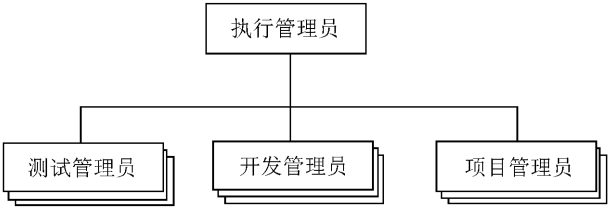


图 11－3 集中管理

这 3 个组织结构只是众多可用结构的简化示例,在软件开发和测试中,一种规格不一定适应所有情况,对一个小组适用,对另一个小组不一定适用,然而,有一些规范可以遵循。下面提供一套选择标准,可作为评估不同方案的基础。在多大程度上该组织结构:

- 提供快速决断能力。
- 提倡合作,尤其是产品开发和测试开发之间的合作。
- 支持一个独立、正规、不带偏见、精干、人员配置合理并有成就的测试机构。
- 帮助协调测试与质量责任之间的平衡。
- 协助在本章前面提到的测试管理要求。
- 为测试技术提供所有权。
- 可利用现有资源,特别是人员配备。
- 对职工(包括经理)的道德和事业产生积极影响。

11.4 能力成熟度模型 CMM

20 世纪 80 年代,卡内基—梅隆大学的软件工程研究所为美国国防部开发了软件过程评估方法和能力成熟度模型 CMM,这是一个机构对自己的软件成熟程度进行评估的模型。软件能力成熟度 CMM 现已成为一个行业标准模型,用来定义和评价软件公司开发过程的成熟度,为提高软件质量提供指导。

11.4.1 CMM 的等级

为了评估开发机构按照现代软件工程方法开发软件的能力,SEI 定义了 5 个过程成熟度等级,被称为能力成熟度模型 CMM:

① 1 级(初始级)。该等级的软件开发过程具有不可预测,难以控制的特征。项目成功依靠个人能力和运气。软件开发过程没有通用的实际计划,难以监视和控制,开发的时间和费用无法预知,测试过程和其他过程混杂在一起。

② 2 级(可重复级)。该等级成熟度的最好描述是可重复以前熟悉的任务。软件开发具有一定的组织性,使用了基本软件测试行为,例如测试计划和测试案例。

③ 3 级(定义明确级)。该等级具备了组织化思想,而不仅仅是针对具体项目。通用管理和工程活动被标准化和文档化。在测试开始之前,审查和检测测试文档和计划,测试人员与开发人

员独立,测试结果用于确定软件发布的时间。

④ 4 级(可控级)。在该成熟度等级中,组织过程得以度量和控制。软件发布时间由事先确定的指标决定(例如,直到每 1 000 行代码只有 0.5 个以下故障时才能发布),软件在没有达到目标之前不能发布。在整个项目开发过程中收集开发过程和软件质量的详细情况,经过调整校正偏差,使项目按计划进行。

⑤ 5 级(优化级)。该等级是第 4 级的不断提高。尝试新的技术和处理过程,评价结果,着重过程改进,以期达到质量更佳的等级。

从当今整个软件公司的现状来看,大多数机构的软件过程开发环境还相当不成熟,根据是否真正采用成熟的软件过程来衡量,全世界大多数软件公司的能力成熟度为 1 级,多数为 2 级,少数为 3 级,极少数为 4 级,能力成熟度为 5 级的软件公司更是凤毛麟角。5 个等级如图 11 – 4 所示,为评测软件公司开发能力成熟度提供了简单的方式。

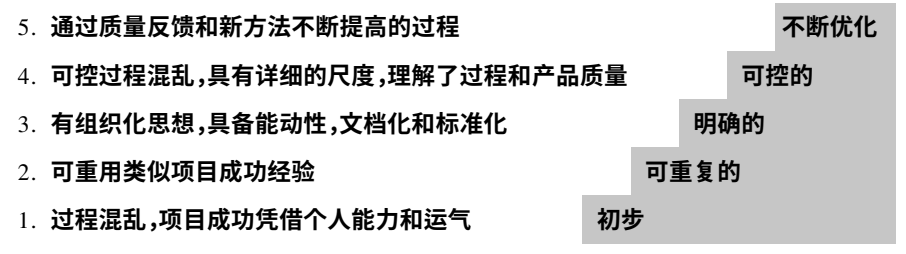


图 11 – 4 CMM 软件成熟度

11.4.2 CMM 等级 3

正确认识软件能力成熟度是测试成功的基础,为此需要对所处的开发环境的成熟程度及其对测试过程的影响做出客观的评价。在实际领域里开展软件测试时,这 1 ~ 5 级就表明了实际工作环境的状况。尽管许多软件测试人员的日常工作并不涉及到软件过程改进和能力成熟度,但 CMM 描述的目标和活动对于致力于软件测试过程改进的人来说具有十分重要的意义。实际上,等级 3 中与测试相关的活动有:

(1) 活动 5,按照项目定义的软件过程开展软件测试

① 顾客和最终用户在适当的时候参与制定和评审测试准则。

② 采用有效的测试方法进行软件测试。

③ 测试的充分性由以下因素决定:

- 测试级别(例如,单元测试、集成测试、系统测试等)。
- 测试策略(例如,黑盒测试、白盒测试)。
- 测试覆盖(例如,语句覆盖,分支覆盖)。

④ 当被测软件或环境发生变化时,在每个相关级别上进行回归测试。

⑤ 在测试计划、测试过程和测试用例准备使用之前,对它们分别进行同行评审。

⑥ 对测试计划、测试过程和测试用例进行管理和控制。

⑦ 当需求、软件设计或代码发生变化时,应适当修改测试计划、测试过程和测试用例。

(2) 活动 6,按照项目定义的软件过程,计划并开展软件的集成测试

① 根据软件开发计划制定集成测试计划,编制相应的文档。

② 负责软件需求、软件设计及系统测试和验收测试的人员,参与评审集成测试计划。

③ 对照指定的软件需求文档和设计文档,开展集成测试。

(3) 活动 7,计划并实施软件的系统测试和验收测试以表明软件满足其需求

① 尽早安排用于软件测试的资源,做好充分的测试准备。

② 在测试计划中包括系统测试和验收测试。最终用户参与评审系统测试和验收测试计划。

③ 由独立于软件开发的测试小组来制定测试计划和测试用例。

④ 对测试用例建立文档,并在测试之前,由最终用户参与评审测试用例。

⑤ 用文档记录测试期间发现的各种问题。

⑥ 对测试结果进行管理和控制。

很显然,测试预算中非技术性(管理)开销的总量与过程成熟度等级成反比。当开发部门处于 1 级时,其开发行为具有很大的不确定性,测试部门根本不能对自己的活动进行计划,于是将更多的时间花在管理问题上而不是技术问题上。

11.5 ISO 9000 标准

与软件质量有关的另一组标准是 ISO 9000。ISO 是一个国际化标准组织,它为小到螺栓、螺母,大到质量管理和质量保证等所有制造行业设立标准。

1. ISO 9000 标准的适用范围

ISO 9000 定义了一套关于质量管理和质量保证的标准,有助于公司一致地交付符合客户质量要求的产品(或者服务)。无论是一家汽车修理公司,还是拥有数亿资金的集团公司,是制作软件、还是配送快餐,ISO 9000 都适用。原因有两个:

- 它的目标在于开发过程,而不是产品。它关心的是进行工作的组织方式,而不是工作成果。它不是设法规定生产线上的小产品或者光盘中软件的质量等级。我们知道,质量是相对的、主观的,公司的目标应该是达到满足客户要求的质量等级。利用质量开发过程可望实现该目标。
- ISO 9000 只决定过程的要求是什么,而不管如何达到。例如,标准声明软件小组应该计划和实施产品设计审查,但是没有说如何才能达到这个要求。实施设计审查是负责任的设计小组应该做的,但是如何组织和执行设计审查完全取决于制造产品的各个小组。ISO 9000 指出要做什么,但不指出应该怎么做。

ISO 9000 标准中针对软件的部分是 ISO 9001 和 ISO 9000 – 3。ISO 9001 针对设计、开发、生产、安装和服务产品,而 ISO 9000 – 3 则针对开发、供应、安装和维护计算机软件。

2. ISO 9000 – 3 标准的主要内容

- 开发详细的质量计划和程序控制配置管理、产品验证和合法性检查等措施。
- 准备软件开发计划,包括项目定义、产品目标清单、进度、产品说明书,如何组织项目的描述,风险和假设的讨论,控制策略等。
- 使用客户容易理解、测试时容易进行合法性检查的用语来表述说明书。
- 计划、开发、编制和实施软件设计审查程序。
- 开发控制软件设计随产品生命周期变化的程序。

- 开发和编制软件测试计划。
- 开发检测软件是否满足客户要求的方法。
- 实施软件合法性检查和验收测试。
- 维护测试结果的记录。
- 解决软件故障的方式。
- 证明产品在发布之前已经就绪。
- 开发控制产品发布过程的程序。
- 明确指出和规定应该收集何种质量信息。
- 使用统计技术分析软件开发过程。
- 使用统计技术评估产品质量。

3. 相关网址

如果有兴趣了解 ISO 9000 标准的详情,可查看以下网址:

- 国际标准化组织(1SO)(网址 www. iso. cn)
- 美国质量协会(ASQ)(网址 www. asq. org)
- 美国国家标准学会(ANSl)(网址 www. ansi. org)

小结

软件测试是为了发现软件故障而执行程序的过程,而软件质量保证的主要职责是检查和评价当前软件开发过程,并设法达到防止软件故障出现的目标。软件测试是提高软件质量的关键步骤,建立软件测试管理体系的主要目的是确保软件测试在软件质量保证中发挥应有的关键作用。测试机构是专门从事测试活动的一种资源或一系列的资源,只有不持偏见的人才能对软件质量提供不持偏见的度量。

软件能力成熟度 CMM 现已成为一个行业标准模型,用来定义和评价软件公司开发过程的成熟度,为提高软件质量提供指导。ISO 是一个国际化标准组织,为所有制造行业设立质量管理和质量保证标准。

第 11 章习题

1. 测试小组和软件质量保证小组的职责一样吗?为什么?
2. CMM 等级与软件测试有关系吗?为什么?
3. 如果某公司在软件开发中引用了 ISO 9000 – 3 标准,那么它的 CMM 等级是多少,为什么?

第 12 章　软件测试职业指导

多年的软件开发实践,人们逐渐认识到软件测试的重要意义。在软件工程学科短暂而又风云变幻的历史中,软件测试人员与开发人员之比发生了巨大的变化,测试人员的比例急剧增加。软件测试人员已成为炙手可热的人才。

本章重点:

- 优秀软件测试工程师应具备的素质
- 软件测试信息资源

12.1　软件测试职位

随着软件规模和复杂性的日益增加,进行专业化高效软件测试的要求也越来越严格,经验丰富的软件测试员已成为炙手可热的人才,下面介绍几种软件测试职位。

(1) 软件测试技术人员

这一般是入门级的测试职位,负责建立测试硬件和软件配置,执行简单的测试脚本或者测试自动化,可能还要分离和再现软件故障,测试技术人员是步入软件测试殿堂的好方法。

(2) 软件测试工程师

大多数公司拥有不同经验和专长的软件测试工程师。初级测试工程师可以履行技术职责,进而执行更高级和复杂的测试。在提高的过程中,可以编写自己的测试案例和测试程序,并参与设计和说明书审查,并分离、再现和报告发现的软件故障。

(3) 软件测试负责人

测试负责人负责软件项目主要部分的测试,有时负责整个小型项目的测试。他们通常为负责的范围制订测试计划,监督其他测试人员实施测试,重点收集产品的使用频度,向管理部门呈报。他们一般不履行软件测试员的职责。

(4) 软件测试管理员

测试管理员监督整个项目甚至多个项目的测试。测试负责人要向他们报告,他们和项目管理员、开发管理员一起设定进度、优先级和目标,为小组测试制定格调和策略,负责为项目提供合适的测试资源——人员、设备、场地等。

12.2　优秀软件测试工程师应具备的素质

因为软件故障,生意失败、致人死命的事太多了,软件测试人员的任务是在它们传播出去之前,高效而专业地找出这些软件故障。

软件测试是一项严谨的工作。一名优秀的软件测试工程师应具备的素质:

(1) 沟通能力

和系统有关的所有人员都处在一种既关心又担心的状态之中。用户担心将来使用一个不符合自己要求的系统,开发者则担心由于用户要求不正确而使他不得不重新开发整个系统,管理部门则担心这个系统突然崩溃而使它的声誉受损。测试者必须能够同测试涉及的所有人进行沟通,要对他们每个人都具有足够的理解和同情,具备了这种能力可以将测试人员与相关人员之间的冲突和对抗减少到最低程度。

(2) 技术能力

就总体言,开发人员对那些不懂技术的人持一种轻视的态度。一旦测试小组的某个成员作出了一个错误的断定,那么他们的可信度就会立刻被传扬了出去。一个测试者必须既明白被测软件系统的概念又要会使用工程中的那些工具,要做到这一点需要有几年以上的编程经验。

(3) 兴趣和自信心

测试者应对软件测试感兴趣,对自己的观点有足够的自信,如果具备了这两点,那么在开发过程中不管遇到什么样的困难,都能克服。

(4) 说服力

软件测试员找出的软件故障有时被认为不重要,不用修复。测试员要善于表达观点,表明软件故障为何必须修复,并通过实际演示力陈观点。如果采取的方法过于强硬,对测试者来说,在以后和开发部门的合作方面就相当于‘ 赢了战争却输了战役 ”。

(5) 幽默感

幽默感可以帮助测试人员和开发小组进行良好的沟通,试着给你的开发小组找一个“ BUG 杀手 ”,或对他们说‘ 我简直不敢相信,你写的程序居然到现在没有找到 BUG ”。在遇到狡辩的情况下,一个幽默的批评将是很有帮助的。

(6) 耐心

一些质量保证工作需要难以置信的耐心。有时你需要花费惊人的时间去分离、识别和排除一个故障。这个工作是那些坐不住的人无法完成的。

(7) 怀疑精神

没有绝对的正确,错误总是难免的,测试人员应具有叛逆心理,别人认为不可能发生的事,他却认为可能发生。测试者可以听每个人的说明,但他必须保持怀疑直到他自己看过以后。

(8) 自我督促

测试工作很容易使你变得懒散。只有那些具有自我督促能力的人才能够使自己每天正常地工作。

(9) 探索精神

软件测试员不会害怕进入陌生环境。他们喜欢拿到新软件,安装在自己的机器上,观看结果。

(10) 故障排除能手

软件测试员善于发现问题的症结,他们喜欢猜谜。

(11) 不懈努力

软件测试员总是不停尝试。他们可能会碰到转瞬即逝或者难以重建的软件故障,他们不会心存侥幸,而是尽一切可能去寻找,不解决誓不罢休。

(12) 创造性

测试显而易见的事实,那不是软件测试员。他们的工作是想出富有创意甚至超常的手段来寻找软件故障。

(13) 追求完美

考虑问题要全面,结合客户的需求、业务的流程和系统的构架等多方面考虑问题,力求完美,尽管有些事情无法做到,但应该去尽力尝试,接近目标。

(14) 判断准确

软件测试员要决定测试内容、测试时间以及看到的问题是否算做真正的故障。

(15) 老练稳重

软件测试员不害怕坏消息。他们必须告诉程序员,你的‘孩子”(程序)很丑。优秀的软件测试员知道怎样老练地处理这些问题,和不够冷静的程序员怎样合作。

(16) 心情愉快

保持一个良好的心情,否则可能无法把测试作好。不要把生活中的不愉快的情绪带到工作中来。

除了这些素质,了解软件是怎样编写的,可以从不同角度找出软件故障,从而使测试更加高效。对于其他行业的专家来说,其专业知识对于开发新产品的软件小组可能富有价值。软件的功能是为了解决现实问题,因此,数学、烹饪、航空、木工、医疗等知识都将对该领域的故障有莫大的帮助。

12.3 软件测试信息资源

软件测试是一项复杂而艰巨的任务。对一个系统进行有效的测试所需要的技能绝对不比进行软件开发需要的少,事实上,测试者可能会遇到许多开发者不可能遇到的问题。要想获得测试成功,需要组织、训练和实践。

12.3.1 正规培训

现在,大多数公司把软件测试视为技术工程专业工作。他们意识到在项目组中培训软件测试员,并在开发过程中早期投入工作可以制造出质量更优的软件。

随着软件测试真正成为一门重要学科,许多大学和学院开设了软件测试课程。许多商业团队和技术院校也开设了软件测试和流行软件测试工具使用的培训班。某些甚至授予软件测试的相关学位和认证。

另一个培训选择是出席专业软件测试会议。我国甚至世界各地每年召开此类会议,提供了聆听来自测试行业讲话的机会。各类资料从最基础的到技术性极高的无所不包。这些会议最重要的部分是与同类软件测试员见面和交谈的机会,交流想法、争论的故事。和解决之道。

下面是一些比较流行的会议:

(1) 全国容错计算学术会议

由中国计算机协会容错专业委员会主办,软件测试作为一个主要的讨论专题,聚集了我国软件测试界一批专家、学者,共同关注我国软件容错技术、软件测试技术、软件安全及测试自动化方面等的进展,每两年召开一次。

(2) 全国测试学术会议

由中国计算机协会容错专业委员会主办,每两年召开一次,主要宗旨是促进软件测试专业人士间的相互交流。

(3) 软件测试分析和审查(STAR)会议,由软件质量工程学会(其网址为 www. sqe. com/star-east 和 www. sqe. com/starwest)主办。STAR 会议的焦点集中在软件测试和软件工程方面,其宗旨是促进和帮助实践者和管理者,使他们在软件测试方面的工作更加富有成效。

(4) 计算机软件测试国际会议和博览会(TCS)

由美国专业开发学会(其网址为 www. uspdi. org)主办。TCS 以国际发言人为特色,展示资深专业人士目前软件测试的最佳方法,提供与各国同仁分享经验和教训的机会,供货商在此展览产品和服务。

(5) 国际质量周

由软件研究公司(www. soft. com)主办。质量周和质量周欧洲会议集中关注软件测试技术、质量控制、风险管理、软件安全性及测试自动化方面的进展。

(6) 国际软件测试会议(ISTC)

由质量保证学会(www. qaiusa. com)主办。ISTC 以来自软件测试和质量保证行业的专家发言为特色,实践者分享他们的测试过程、测试管理和度量策略以及最有效的测试技术,测试工具销售展览与会议同时进行,使与会者能看到展示的工具,并和工具支持人员进行面对面的交流。主题讨论范围从基本软件测试到测试自动化到测试最新技术。

(7) 软件质量国际会议(ICSQ)

由美国质量委员会软件分会(www. asq - software. org)主办。与其他会议一样,ICSQ 提供与其他软件测试和质量保证专业人士交流想法和方法的机会。

(8) 软件测试国际会议(ICSTEST)

由软件质量系统学会(www,icstest. com)主办。ICSTEST 是在德国举办的国际测试会议,这是关于软件测试方面展示、指导、讨论和交流经验的论坛。

(9) 软件质量第二世界会议(2WCSQ)

2WCSQ 是关于软件质量的世界范围会议,成员来自 27 个不同国家。2000 年会议在日本举行。

12.3.2 因特网

因特网上拥有关于软件测试的丰富信息。虽然搜索“ software testing ”或者“ software test ”可以找到一些资料,但下列专门针对软件测试和软件故障的流行网站可以提供一些入门引导。

- 测试时代网 :http ://www. testage. net/,开辟有测试论坛和测试时代专栏,并提供有关软件测试的经验交流及测试工具下载。如图 12 - 1 所示为其界面图。
- 软件工程专家网 :http ://www. 51cmm. com/SoftTesting/,设有软件测试专栏,介绍了许多与软件测试相关的文章和测试工具。
- 赛迪网 :http ://tech. ccidnet. com/pub/column/c319. html,给出了许多与软件测试相关的文章和测试经验。
- Software Testing Hotlist (www. io. com/ ~ wazmo/qa)列出了一些与软件测试相关的网站和

图 12 – 1 “ 测试时代网 ”界面

文章。

• Software Testing Online Resources（www. mtsu. edu/ ~ storm）旨在成为软件测试研究者和从业者的网上第一站。

• Comp. software. testing 新闻组提供测试员和测试管理员关于工具、技术和项目的即时讨论。

• Bug Net（www. bugnet. com）公布在商业软件中发现的软件故障，并指出相应的修复措施。

• QA Forums（www. qafomms. com）提供软件测试、自动化测试、测试管理、测试工具等主题的即时讨论。

• Comp. risks 新闻组描述和分析近期软件失败。

12.3.3 专业组织

专门针对软件、软件测试和软件质量保证的一些非盈利性专业组织有：

• 计算机学会（ACM）在 www. acm. org 及其软件工程专门兴趣团队（SIGSOFT）在 www. acm，. org/sigsoft 拥有教育和科学计算方面的 8 000 多个会员。

- 美国质量委员会(ASQ)在 www. asq. org 及其软件分会在 www. asq – software. org 每年 10 月份国际质量月主办国际质量论坛。他们发行质量方面的刊物和文章,并管理质量认证工程师(CQE)和软件质量认证工程师(CSQE)的任命。
- 软件质量委员会(SSQ)在 www. ssq. org 中,将其目标定为“成为那些把‘质量’作为软件最大目标的人们的公认论坛”。

小结

软件测试是一项复杂而艰巨的任务。软件测试人员的任务是高效而专业地找出软件中存在的故障。要想成为一名优秀的软件测试工程师,不仅需要树立正确的测试态度和足够的自信心,还需要经过训练和实践,在实践中不断积累测试经验。

第 12 章习题

1. 软件测试人员的目标是什么?
2. 在因特网上查找软件测试职位时,应使用什么样的关键字进行搜索?

附录　软件工程的测试标准

一、主要软件测试标准

IEEE/ANSI Standard 829 – 1983

IEEE Standard for Software Test Documentation（Reaff. 1991）

该标准定义了 8 份覆盖整个测试过程文件的内容和格式。测试计划规定了测试活动的范围、方法、资源和进度。定义了测试项目、测试任务、负责各任务的人员以及与计划有关的风险。该标准显示了这些文件制定时的相互关系以及它们与测试过程的关系。

IEEE/ANSI Standard 1008 – 1987

IEEE Standard for Software Unit Testing（Reaff. 1993）

软件单元测试是一个过程，包括测试计划的制定、测试软件的开发以及按要求对测试单元进行度量也就是将样本数据施加到单元上，并将单元的实际性能与单元需求文件中规定的要求进行比较。

该标准定义了一种系统且文档化的单元测试集成方法。该方法采用了单元设计、单元实施信息以及单元需求，以便确定测试的完整性。该标准描述了一个由阶段、活动和任务等层级组成的测试过程。此外，它还定义了每一活动的最起码的任务数量，当然每一活动的任务是可以增加的。

二、其他与软件测试有关的标准

IEEE/ANSI Standard 1012 – 1986

IEEE Standard for Software Verification and Validation Plans（Reaff. 1992）

该标准有 3 个方面的用途：

① 为关键和非关键软件的软件验证和确认计划（SVVP），提供有关格式和内容方面的统一和基本的要求。

② 为关键软件定义最基本的验证和确认（V&V）任务，以及按要求必须包括在 SVVP 之内的输入和输出。

③ 推荐任选的 V&V 任务，用于就特定的 V&V 工作对 SVVP 进行合理的修改。

IEEE/ANSI Standard 1028 – 1988

IEEE Standard for Software Reviews and Audits

软件评审和审计是软件开发周期中软件产品评估的基本组成部分。该标准对评审或审计人员的评估行为提供指南。包括对关键和非关键软件都适用的过程，以及对评审和审计的执行所要求的步骤。

IEEE/ANSI Standard 730 – 1989

1EEE Standard for Software Quality Assurance Plans

该标准将法律责任作为理论基础，面向关键软件的开发和维护，在这方面如果出现故障会影响安全或引起巨大的经济或社会损失。主要目标是描述特定项目所有的计划和系统行为，以便

使人确信该软件产品符合现有的技术要求。

该标准为软件质量保证计划规定了格式和基本内容。

IEEE/ANSI Standard 610.12－1990

Standard Glossary Of Software Engineering Terminology

该标准是 IEEE Standard 729 的修改和重新规定。该标准包含 1 000 多条术语的定义，构成了软件工程的基本词汇。它以美国国家标准学会（ANSl）和国际标准化组织（1SO）的术语为基础，提倡软件工程和相关领域词汇的准确性和一致性。

其他的软件工程标准

IEEE/ANSI Standard 828－1990，

IEEE Standard for Software Confiquration Manaqement Plans

该标准在格式上类似于 IEEE Standard 730，但限于处理软件配置管理问题。该标准确定了配置标识、配置控制、配置状态注册和报告以及配置审计和评论方面的要求。这些要求的实施提供了一种记录、交流并控制软件产品项目演变的手段。它为软件产品项目在其开发和维护生存周期的进化过程中的完整性和连贯性提供了保证。

IEEE/ANSI Standard 830－1993

IEEE Recommended Practice for Software Requirements Specifications

该指南描述了软件需求规格说明方面的一些优秀方案。为使读者作出正确选择，该标准还提供了大量的辅导材料。该指南包括了优秀软件需求规格说明的特征，以及规格说明的方法和相关格式。

IEEE/ANSI Standard 982.1－1988

IEEE Standard Dictionary of Measures to Produce Reliable Software

该标准为所选度量提供了定义。这些度量以生产出可靠的软件为目的，适用于整个软件开发生存周期。该标准没有指定或要求采用任何一种度量，目的是描述各个度量和它们的用途。

该标准的补充标准是 982.2，它为 IEEE Standard 982.1 中度量的使用提供指南。它为该行业提供所需的信息，以便充分利用 IEEE Standard 982.1。

IEEE/ANSI Standard 990－1987

IEEE Recommended Practice for Ada as a Program Design Language（Reaff. 1992）

该推荐实际为基于 Ada 编程语言句法和语义的程序设计语言（PDL）特征提供了许多建议，这些建议反映了各种最新的优秀方案。在该文件中，它们称为 Ada PDL。

IEEE/ANSI Standard 1002－1987

IEEE Standard Taxonomy for Software Engineering Standards（Reaff. 1992）

该标准描述软件工程标准分类的形式和内容。解释了各种类型的软件工程标准，它们的功能和外部关系以及各种功能在软件生存周期中所起的作用。组织可以采用该分类作为标准的开发或评估计划的方法之一。也可以作为标准分类或编制标准手册的基础。

IEEE/ANSI Standard 1016－1987

IEEE Recommended Practice for Software Design Descriptions（Reaff. 1993）

软件设计描述是对软件系统的描绘，可用作交流软件设计信息的媒介。该媒介介绍了软件设计的文档编制。它为软件设计描述指定了必要的信息内容和推荐的组织。

IEEE/ANSI Standard 1042 – 1987

IEEE Guide to Software Configuration Management (Reaff. 1993)

该指南的用途是为与 IEEE Standard 828 兼容的软件配置管理(SCM)实践的计划提供指导。该指南的重点是 SCM 计划的过程,同时也为软件配置管理的理解提供了广泛的视角。

IEEE/ANSI Standard 1058. 1 – 1987

IEEE Standard for Software Project Management Plans(Reaff. 1993)

该标准指定软件工程管理计划的格式和内容。它没有指定开发或项目管理计划应采用的过程或技术,也没有提供项目管理计划的例子。相反,该标准为组织建立自己的开发项目管理计划的方法和过程提供了基础。

参 考 文 献

1 Abramson D, Sosic R. A Debugging and Testing Tool for Supporting Software Evolution. Automated Software Engineering, 1996, 3: 369 ~ 390

2 Beizer B. Software Testing Techniques, Second Edition. Van Nostrand Reinhold Company, 1990

3 Binder R. Object – Oriented Software Testing. Communications of the ACM. Guest Editor, 1994, 37(9): 28 ~ 29

4 Chen H Y, Tse T H, Chan F T. et al. In: Black and White. An Integrated Approach to Class – Level Testing of Object – Oriented Programs. ACM Transactions on Software Engineering and Methodology. 1998, 7(3): 250 ~ 295

5 Clarke L. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering, 1976, SE – 2(3): 215 ~ 222

6 Doong R K, Frankl P. The ASTOOT Approach to Testing Object – Oriented Program. ACM Transactions on Software Engineering Methodology, 1994, 3(2): 101 ~ 130

7 Kit E. Software Testing In the Real World: Improving the Process. Pearson Education Limited, 1995

8 Ferguson R, Korel B. The Chaining Approach for Software Test Data Generation. ACM Transactions on Software Engineering and Methodology. 1996, 5(1): 63 ~ 86

9 Fiedler S P. Object – Oriented Unit Testing. Hewlett – Packard Journal. 1989(4): 69 ~ 74

10 Frankl P G, Weyuker E J. An Applicable Family of Data Flow Testing Criteria. IEEE Transactions on Software Engineering. 1988, 14(10): 1483 ~ 1497

11 Goodenough J B, Gerhart S L. Toward a Theory of Test Data Selection. IEEE Transactions on Software Engineering, 1975, SE – 1

12 Hall P V. Relationship between Specification and Testing. Information and Software Technology. 1991. 33(1): 47 ~ 52

13 Horgan R, Mathur A. Handbook of software reliability engineering. In: Software Testing and Reliability. CA: IEEE Computer Society Press, 1996

14 Howden W E. Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering. 1976, SE – 2(3): 208 ~ 215

15 Howden W E. Symbolic Testing and the DISSECT Symbolic Evaluation System. IEEE Transactions on Software Engineering. 1977, SE – 3(4): 266 ~ 278

16 Howden W E. Functional Testing. IEEE Transactions on Software Engineering. 1980, SE – 6(2): 162 ~ 169

17 Howden W E. Weak Mutation Testing and Completeness of Test Sets. IEEE Transactions on Software Engineering. 1982, 8(4): 371 ~ 379

18 Huang J C. Program Instrumentation and Software Testing. Computer. 1978

19 Jeng B , Weyuker E J. A simplified domain – testing strategy. ACM Transactions on Software Engineering and Methodology. 1994. (3) :254 ~ 270
20 Jorgensen P C , Erickson C. Object – Oriented Integration Testing. Communications of the ACM. 1994 :30 ~ 38
21 Kaner C. Falk J , Nguyen H Q. Testing Computer Software. Second Edition. John Wiley & Sons Inc ,1999
22 Korel B. Automated Software Test Data Generation. IEEE Transaction on Software Engineering. 1990 , 16(8) :870 ~ 879
23 Kung D, Hsia P , Gao J. Testing Object – Oriented Software. IEEE Computer Society , 1998
24 Laski J , Korel B. A Data Flow Oriented Program Testing Strategy. IEEE Transaction on Software Engineering. 1983(9) :33 ~ 43
25 Lyu M R. Editor, Handbook of software reliability engineering. CA :IEEE Computer Society Press ,1996
26 Marick B. The Craft of Software Testing. NJ :PTR Prentice Hall ,1995
27 Myers G J. Software Reliability :Principles & Practices. New York :John Wiley & Sons , 1976
28 Myers G J. The Art of Software Testing. New York :John Wiley & Sons ,1979
29 Jorgensen P C. Software Testing :A Craftsman's Approach. Second Edition. CRC Press ,2002
30 Perry W E. Effective Methods for Software Testing. Second Edition. John Wiley & Sons. 2000
31 Poston R M. A Guided Tour of Software Testing Tools. Aonix Corp ,1998
32 Rapps S, Weyuker E J. Selecting software test data using data flow information. IEEE Transactions on Software Engineering. 1985 ,SE – 11(4) :367 ~ 375
33 Patton R. Software Testing. Sams Publishing ,2001
34 Smith M D, Robson D J. A Framework for Testing Object – Oriented Programming. Journal of Object – Oriented Programming , 1992 , 5(3) :45 ~ 53
35 Weyuker E J. The Evaluation of Program – Based Software Test Data Adequacy Criteria. Communications of ACM. 1988 , 31(6) :668 ~ 675
36 Weyuker E J. The Cost of Data Flow Testing :An Empirical Study. IEEE Transactions on Software Engineering. 1990 ,16(2) :121 ~ 128
37 Weyuker E J, Goradia T, Singh A. Automatically Generating Test Case from a Boolean Specification. IEEE Transactions on Software Engineering. 1994 , 20(5) :353 ~ 363
38 White L J , Cohen E I. A domain strategy for computer program testing. IEEE Transactions on Software Engineering. 1980 , SE – 6(3) :247 ~ 257
39 Zhu H , Hall P. Test Data Adequacy Measurement. Software Engineer Journal. 1993 , 8(1) :21 ~ 30
40 Zhu H. A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria. IEEE Transactions on Software Engineering. 1996 , 22(4) :248 ~ 255
41 **北京航空航天大学软件工程研究所**. C ++ **软件分析与测试工具** SafePro/C ++ **用户册**. 2000
42 **胡谋. 容错技术. 北京 :中国铁道出版社**,1995

43 徐仁佐. 软件可靠性模型及应用. 北京 :清华大学出版社,1994
44 张海藩. 软件工程导论. 北京 :清华大学出版社,1998
45 郑人杰. 计算机软件测试技术. 北京 :清华大学出版社,1992
46 赵瑞莲. 软件测试方法研究. 中国科学院计算技术研究所博士论文. 2001
47 赵瑞莲,闵应骅. 基于谓词切片的字符串测试数据自动生成. 计算机研究与发展,2002 年第 4 期.
48 赵瑞莲,闵应骅. 一种基于规范和程序域分析的软件测试方法. 计算机研究与发展,2003(6)
49 朱鸿,金凌紫. 软件质量保障与测试. 北京 :科学出版社,1997

参 考 网 站

http ://www. testage. net/
http ://www. 51cmm. com/SoftTesting/
http ://tech. ccidnet. com/pub/column/c319. html
http ://www. parasoft. com/